ACTING, PLANNING, AND LEARNING

Malik Ghallab
LAAS-CNRS, University of Toulouse, France

Dana Nau
University of Maryland, USA

Paolo Traverso
FBK, Trento, Italy

This material will be published by Cambridge University Press as
Acting, Planning, and Learning by Malik Ghallab, Dana Nau, and Paolo Traverso.
This pre-publication version is free to view and download for personal use only.
Not for redistribution, resale, or use in derivative works.
© 2024 by Malik Ghallab, Dana Nau, and Paolo Traverso.

October 9, 2024

*To Janette, Lise, Elena, and to all our family members
who remained supportive during a long project
that consumed much of our time and attention.*

Contents

Free pre-publication, for personal use only. To be published by Cambridge University Press.

Foreword

Over the past decade, Artificial Intelligence (AI) has made remarkable breakthroughs, particularly in the realm of deep learning and foundation models —sub-symbolic machine learning approaches that leverages deep neural networks with hundreds of billions parameters. These models are often called black boxes as their human interpretation and understanding is very limited. This technology has been instrumental in enhancing interaction, perception, and natural language processing, sometimes even surpassing human capabilities. As a result, some researchers have begun to equate AI with deep learning and foundation models. However, I believe this is a significant misconception.

AI encompasses far more than just sub-symbolic machine learning; it includes symbolic (i.e., human-understandable) modeling, search algorithms, and reasoning techniques - all vital aspects of human intelligence that extend beyond machine learning, and can potentially utilize it to enhance algorithm performance and model accuracy.

Planning and acting are intrinsic human abilities. Even young children naturally plan and act, learning from the consequences of their actions in an environment and refining their skills as they grow. Machines have not yet reached human-level proficiency in planning and acting, as well as in their integration with learning, leaving considerable room for advancement and improvements in autonomous intelligent systems.

This book serves as a crucial milestone in the study of planning, acting, and learning, exploring how these intelligent features can be effectively combined and integrated to improve the performance of intelligent systems. The authors, Malik Ghallab, Dana Nau, and Paolo Traverso, are three outstanding scientists and researchers who have achieved significant recognition and visibility within the AI international scientific community. This is the third book they have written on the subject: the first focused on planning, while the second explored the interaction between acting and planning. This third book marks an important step forward by also addressing the intersection of acting, planning, and learning. It discusses Deterministic State-Transitions, Hierarchical Task Networks, Probabilistic, Non-deterministic, Hierarchical-Refinement, and Temporal Models, while also considering Robotic Motion and Manipulation. Additionally, it explores the emerging capabilities of Large Language Models and how they can be applied in this field, a very recent and relevant topic at the intersection between sub-symbolic and symbolic AI.

The book is not only a valuable reference for scientists working in the area but also serves as a textbook for graduate students, offering a clear, comprehensive, and well-organized catalogue of techniques and algorithms for domain modeling, plan construction, and execution, as well as the integration of learning in all these

Free pre-publication, for personal use only. To be published by Cambridge University Press.

activities. I have no doubt that I will recommend it in my courses and use it as a personal reference.

Prof. Michela Milano
University of Bologna

Preface

For an agent to act intelligently, three essential cognitive functions are acting, planning, and learning. This book is about ways to automate and integrate them. It is a successor to our previous books on automated planning [410] and on combining planning and acting [411]. It includes research advances that have occurred since those books were published.

This book covers several types of models, approaches, and algorithms—deterministic, probabilistic, hierarchical, nondeterministic, temporal, and spatial—and discusses how to use them for acting, planning and learning. The published literature on these topics is huge and covers several disconnected areas, not all of which can be covered in a single book. Thus our choice of material was motivated by putting the integration of acting, planning and learning at the forefront.

The book comprises 24 chapters. After Chapter 1, the Introduction, the other chapters are organized into eight parts. The first seven focus on the following representational models, with each part containing chapters on acting, planning, and learning with the given model:

- Part I uses a "classical" deterministic state-transition model, represented using state variables. Several of the concepts in this chapter are used throughout the book.
- Part II adds hierarchical task networks (HTNs) to the state-transition model in Part I.
- Parts III and IV extend the state-transition model in Part I to include, respectively, probabilities and nondeterminism.
- Part V describes a hierarchical refinement approach that builds on the HTN concepts in Part II and the probabilistic model in Part III.
- Part VI models time and concurrency using a chronicle representation.
- Part VII introduces models of robotic motion and manipulation and their combination with more abstract tasks.

Finally, Part VIII includes two chapters on some other important topics that are not within our main focus: large language models, and sensing, monitoring, and goal reasoning.

Using This Book

This book is intended both as an information source for scientists and professionals and as a graduate-level textbook. In most of the chapters, the references are postponed to a discussion section at the end of the chapter. Most of the discussion sections are

Free pre-publication, for personal use only. To be published by Cambridge University Press.

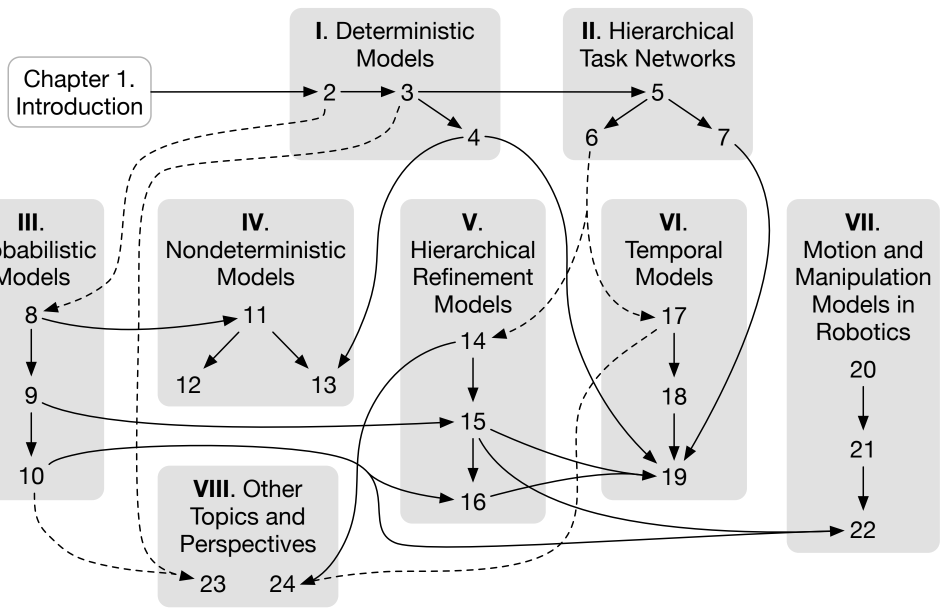

Chapter 1.
Introduction
I. Deterministic Models
2
3
4
II. Hierarchical Task Networks
5
6
7
III. Probabilistic Models
8
9
10
IV. Nondeterministic Models
11
12
13
V. Hierarchical Refinement Models
14
15
16
VI. Temporal Models
17
18
19
VII. Motion and Manipulation Models in Robotics
20
21
22
VIII. Other Topics and Perspectives
23
24

Figure 1. Dependencies among chapters. Each gray box represents one of the book's parts, and the numbers within each part refer to chapters. A solid line means that some (though often not all) of the information in one chapter is needed to understand another chapter. A dashed line means that the information is not required but may be helpful.

followed by sets of exercises. We will make lecture slides and other auxiliary materials available online.[1]

In the pseudocode for our algorithms, all variables are local unless declared global. We assume readers are familiar with the basic concepts of algorithms and data structures at the level of an undergraduate-level computer science curriculum. Two appendices provide information about some mathematical and technical topics that go beyond this background.

In addition to providing a coherent synthesis of the state of the art, this book contains a substantial amount of new material, most of which is presented in comprehensive detail consistent with textbook use. Some sections contain new material that has not yet been implemented and empirically assessed, to provide an invitation for further research.

The study of this book may follow several paths, depending on the reader's needs and familiarity with the material. Figure 1 shows which chapters depend on which others. We hope this will help readers and teachers plan a fruitful journey through the book.

Acknowledgments

We are thankful to Sylvie Thiébaux at LAAS-CNRS, Toulouse, and the Australian National University, and Marcel Steinmetz at LAAS-CNRS, for contributing to Sec-

[1]http://www.laas.fr/planning

tion 9.3 and writing Section 9.4. We are grateful to Michela Milano at the University of Bologna, for writing the Foreword.

We thank several friends and colleagues for their valuable feedback on parts of this book. These include Pascal Bercher, Janette Cardoso, Mark “Mak” Roberts, Luciano Serafini, Oliviero Stock, Fabio Pianesi, Bernardo Magnini, Sylvie Thiébaux, and Silvano Dal Zilio. Dana Nau thanks the students who took a course from a rough draft of this book. Paolo Traverso owes special thanks to Luciano Serafini for his contributions to the chapters on learning deterministic and nondeterministic action schema.

We acknowledge the support of our respective organizations, which provided the support and facilities that helped to make this work possible: LAAS-CNRS and the University of Toulouse, France, the University of Maryland, and FBK in Trento, Italy. Dana Nau thanks ONR, NRL, AFRL, and AFOSR for their support of his work. Malik Ghallab and Paolo Traverso acknowledge support from the AIPlan4EU project (Grant agreement ID: 101016442), and Paolo Traverso acknowledges support from the Future AI Research project (FAIR, PE00000013) under the NRRP MUR program funded by the NextGenerationEU.

The authors declares that no sentence or figure of this book were produced with the help of a generative AI software. They have no conflict of interest regarding the presented methods and techniques, nor with any person or company deploying their machine implementations.

About the Authors

Malik Ghallab is Directeur de Recherche Emeritus at CNRS and the University of Toulouse. His research is focused on AI and Robotics. He contributed to topics such as knowledge representation and reasoning, planning, and learning of skills and models of behaviors. He (co-)authored over 200 scientific papers and several books. He taught AI at several universities in France and abroad, and advised 32 PhDs. He led several AI research programs in France, was director of LAAS-CNRS and CTO of INRIA. He is involved in initiatives regarding socially responsible research in AI and computational sciences. He is a EurAI Fellow, and Docteur Honoris Causa of the University of Linköping, Sweden.

Dana Nau is a Professor Emeritus at the University of Maryland, in the Computer Science Department and the Institute for Systems Research. He has more than 400 refereed technical publications. Some of his accomplishments include the discovery of "pathological" game trees in which looking farther ahead produces worse decisions, the game-playing algorithm used in the program that won the 1997 world computer bridge championship, applications of AI planning in automated manufacturing, AI planning systems such as SHOP, SHOP2, Pyhop, and GTPyhop, and evolutionary game-theoretic studies of factors that affect human behavioral norms. He is an AAAI Fellow, ACM Fellow, and AAAS Fellow.

Paolo Traverso is the Director of Strategic Planning at Fondazione Bruno Kessler (FBK), Trento, Italy. His main research interests are in automated planning and learning under uncertainty. He directed the FBK ICT Research Center from 2007 to 2020, a center of more than 400 people. In 2017, he was appointed Chair of the Strategic Committee of EIT Digital. Since 2019, he has been a member of the Scientific Advisory Board of DFKI. He served as a member of the Board of Directors of AIxIA and of the National Lab on AI and Intelligent Systems. Since 2023, he has been the leader of the National Project on Integrative AI of "FAIR - Future Artificial Intelligence Research". He is the author and co-author of more than one hundred scientific articles. He is an EurAI Fellow, AAIA Fellow, and AIIA Fellow.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

1 Introduction

The ability to act autonomously in the environment is a key feature of intelligence. An AI acting system, for short an *actor*, is a computational artifact capable of autonomous operation in its environment. It can be a software system, such as a web-based service agent, or a robot embodied with sensory-motor devices. Actors require essential cognitive functions without which intelligence is hardly conceivable, and this book focuses on the functions of acting, planning, and learning:

- *Acting* is more than just the sensory-motor execution of low-level commands. There always is a need to decide *how* to perform each task, given the context, and adapt online to changes in the environment.
- *Planning* involves choosing and organizing actions that can achieve a task or a goal. It usually involves reasoning on abstract models of a repertoire of actions the actor may perform.
- *Learning* is critical for acquiring knowledge about actions' actual effects, which actions to perform when, and how to perform and plan them. Conversely, acting and planning can be used to aid learning.

Combining these cognitive functions will be very important for the future of AI. To explain why, we briefly summarize some recent developments in AI research.

During the past few decades, AI has produced numerous success stories. However, most of them were costly, requiring huge development, modeling and adaptation to their respective domains. They also tended to be brittle and narrow, with capabilities that were difficult to extend.[1] For many years, AI learning systems lacked a capability to adapt, generalize, and transfer to other domains. These adaptation capabilities, essential to intelligence, are beginning to be reached in two primary areas: data interpretation and data generation.[2]

- *Data interpretation.* Multi-layered neural networks have extended known principles to provide robust universal approximation classifiers. Moreover, they have incidentally provided, at several abstraction levels, representation features adapted to specific training data. For decades, the field of pattern recognition has devoted significant effort to design representation features characterizing the data at hand. These features are now given for free as latent variables in the successive hidden layers of a neural net. They result from scalable training procedures, thanks to improvements in hardware performances and architectures. AI for data interpretation is no longer costly and narrow. It is

[1] An example is the Watson system [356], the impressive champion of the Jeopardy Q/A game, which was transposed to the medical domain, but not successfully deployed despite huge investments.

[2] This oversimplifies a rich story. See, for example, [725, 758].

Free pre-publication, for personal use only. To be published by Cambridge University Press.

widely deployed for the analysis of all kind of multi-modal data in numerous demanding applications, from astronomy to health and education.

- *Data generation.* Also here the principles have been known for a while: learn an adequate distribution for a domain, and sample from it for a given context. Generative sampling and prediction of the next term in a sequence have benefited from the progress in multi-layered networks in performances and architectures. The recent multi-head attention transformer architecture of Large Language Models, and their extensions in Multimodal Foundation Models, have led to impressive performance in natural language processing and image generation tasks. They also demonstrate emergent but still fragile capabilities in other unexpected common-sense and reasoning tasks. Scalable AI tools for generating texts, images, videos, and sounds are now widely deployed.

From data to actions. This, we believe, is the next big, two-sided challenge for the field. On the one hand, AI has to pursue and leverage on its successful achievements in order to transform current techniques for acting and planning into easily learned and scalable approaches. An actor should be able to extensively and efficiently learn how to act and how to plan. It should also be able to act and plan in order to better learn and adapt to its environment and mission. On the other hand, the challenge is to "put acting into AI." For example, the successful data interpretation and generation methods require numerous actions, such as to gather and select training data, choose meta-parameters, etc. These should be part of the actions learned, planned for and performed by the autonomous agent.

In two previous books, we wrote about automated planning [410] and about combining planning and acting [411]. The interactions among the acting, planning, and learning functions open essential perspectives for addressing the next big AI challenge. We hope through this book to contribute to the education and training of researchers and practitioners tackling this challenge.

The rest of this chapter is organized as follows. Section 1.1 presents a conceptual view of an AI actor, its architecture, and main components. Section 1.2 introduces the types of models needed for the design of an actor. Section 1.3 expresses our concerns and recommendations about important ethical issues associated with autonomous actors. The outline and organization of the book are detailed in the preface.

1.1 Architecture and Components of an Actor

This section introduces the main components and organization of a deliberative actor. It first presents a simplified, conceptual of view an actor's architecture. It then discusses the acting, planning, and learning functions and their interplay.

1.1.1 Architecture

The methods discussed in this book are relevant both for software actors and for actors embodied with sensory-motor devices. The latter are further detailed in Part VII on motion and manipulation. A simplified view of an actor distinguishes two main parts: a deliberation part and an execution platform (see Figure 1.1).

The *execution platform* informs the actor about its environment and its current state. It transforms its commands into actuations that perform its actions (e.g., movements of a limb or of a virtual character). The platform of an embodied actor assembles sensors, actuators and signal processing functions. The actor has to control its platform (e.g., where to put and how to use its sensors and actuators). Hence, it needs a model of the platform's capabilities and limitations.

The *deliberation functions* are used to choose what to do and how to do it to achieve the actor's mission, how to react to changes in the environment, and how to interact with other actors, including humans. We focus the book on the acting, planning, and learning functions. Other functions, namely perceiving, monitoring and goal reasoning, are briefly covers in Chapter 24. Communication, adaptation to, and interaction with other actors are also important. They are not developed *per se*, but Chapter 23 introduces Large Language Models and discusses their possible use as deliberation functions.

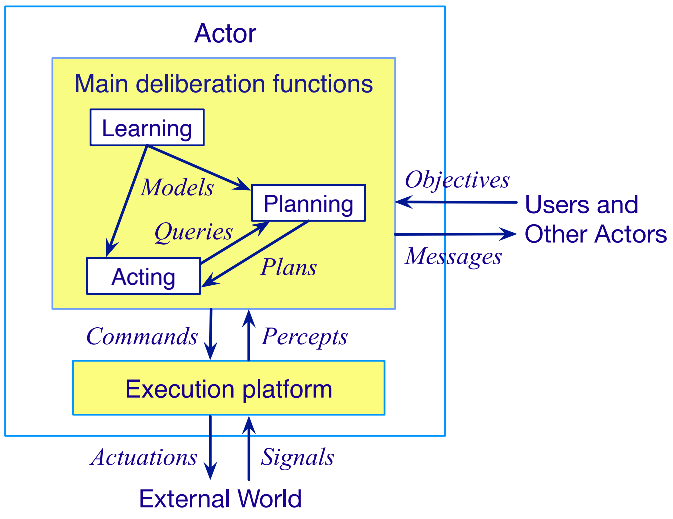


Figure 1.1. Conceptual architecture of an actor.

The architecture depicted in Figure 1.1 is a simplified conceptual schema that can be adapted to different classes of environments and actors. It presents the actor as centralized system, while it can also be distributed. More importantly there are two essential features, implicit in this figure:

- *Hierarchical processing within and across functions.* From abstract tasks to detailed actuations, a hierarchy of methods reduce the complexity of deliberation, and integrate heterogeneous representations and models.
- *Continual online closed-loop adaptation.* The actor predicts what is expected, monitors what is taking place, reacts to events, extends, updates, and repairs its plan, and possibly revises its goals on the basis of its perception and deliberation.

These organizational principles provide a guideline to be adapted to different classes of environments and actors, about which the various parts of the book make different assumptions. Let us now discuss the main components of this architecture.

1.1.2 Planning

Planning is about *what* to do. It relies on a predictive model to foresee what may happen if some actions are performed, and a search over alternative options. It seeks to synthesize a plan, i.e., an organized set of actions that may lead, according to predictions, to a desired goal.

Planning problems vary in the kinds of actions, predictive models and desired plans. In some cases, specialized planning methods can be used with specific problem representations. For instance, motion planning synthesizes a kinematic and dynamic trajectory for moving a device; perception planning generates sensing and interpretation actions to sense the world, recognize or model an object or a scene.

In many cases, there are commonalities to various planning problems. Domain-independent planning tries to grasp these commonalities with abstract models. Domain-independent planners reason about actions by representing them uniformly as state-transformation operators over widely applicable representations of states as relations among objects.

Domain-independent and specialized planning are complementary. In a hierarchically organized actor, an abstract level can be tackled with domain-independent techniques, whereas lower levels may require specialized techniques. The integration of domain-independent and specialized planners raises several challenges, which are exemplified by the integrated task and motion planning problems in Part VII.

1.1.3 Acting

Acting is about *how* to do chosen actions while reacting, in a closed loop, to the observed context in which the activity takes place. An action is considered as a task to be progressively refined, given the current context, into more primitive actions and concrete commands. Whereas planning is a search over *predicted* states, acting is a continual assessment of the current *observed* state, and a consequent adaptation. Acting requires reacting to unexpected changes and exogenous events, which are independent from the actor’s activity. It also requires a correct mapping between what is perceived and actuated and what is reasoned about for acting.

The techniques used in planning and acting can be compared as follows. Planning is organized as an *open-loop* search, a look-ahead process based on predictions. Acting is a *closed-loop* process, with feedback from observed effects and events used as input for subsequent decisions. Domain-independent planners can be developed to take advantage of commonalities among different forms of planning problems, but this is less true for acting systems, which require specific methods.

1.1.4 Interleaving Acting and Planning

Relationships between acting and planning are more complex than a simple linear sequence $\langle plan, act \rangle$. Seeking a complete plan before starting to act is not always feasible, desirable or needed. It is feasible when the environment is predictable and well modeled, as in a manufacturing production line. It is needed in domains with high costs or risks, or when actions are not reversible. In such domains, one often has to engineer the environment to reduce diversity as much as possible beyond what is modeled and can be predicted.

In open, dynamic domains with exogenous events that are difficult to model and fully predict, plans are expected to fail. They cannot be carried out blindly until the end. Plan modification and replanning are part of a global closed-loop process for acting. Replanning is normal and should be embedded in the design of an actor. Metaphorically, planning sheds light on the road ahead, but does not lay an iron rail all the way to the goal.

The interplay between acting and planning can be organized in many ways, depending on how easy it is to plan, how predictable and dynamic the environment is, and how costly or risky the actions are. A general paradigm is the *receding-horizon* model of interleaved planning and acting. It consists of repeating the two following two steps until the goal is reached:

1. Plan from the current state toward the goal, but not necessarily all the way to the goal, stopping at an arbitrary cutoff point called the *planning horizon*.
2. Perform one or more actions of the synthesized plan. Observe the current state and decide whether further planning is needed.

A receding-horizon scheme can have various instantiations. Options depend, for example, on the planning horizon, on what triggers replanning, on the number of actions performed after a planning stage, and whether planning can be interrupted. Furthermore, the planning and acting procedures can be run either sequentially, or in parallel with synchronization. A receding-horizon approach can scale up to large state spaces, and can redirect the planning in a closed loop according to the results of acting. But it may also lead to situations from which the goal cannot be reached.

Depending on the planning horizon, the actor may execute each action as soon as it is planned or wait until a dynamically chosen planning horizon is reached. One should expect the observed state to differ from the predicted one, and to evolve even if no action is executed. This may invalidate a plan and require replanning.

Interleaving acting and planning remains relevant if the planner synthesizes alternative courses of action for different contingencies (see Parts III and IV). It may not be worthwhile or even feasible to plan for all possible contingencies, or the planner may not know in advance what all of them are.

Several instances of the receding-horizon scheme will be illustrated throughout the book, including anytime approaches.

1.1.5 Learning

Learning is a very broad notion that includes many cognitive capabilities. An actor learns if it improves its performance with more autonomy and versatility, including ways to perform new tasks, and adaptation to new or changing environments. Learning may rely on the actor's experiences, instructions from a tutor, and/or data and knowledge gathered from external sources.

Learning alleviates the costly efforts of programming an actor and specifying its environment. Even when such programming can be performed, it can hardly cover all the situations the actor may face, so adaptation by learning provides a significant advantage. Furthermore, learning allows an actor to acquire skills for which the designer may not have formalized knowledge or are difficult to program.[3]

An actor may want to learn a reactive function giving how to act in each situation and context, without further need of reasoning. Alternatively, it may want to learn models with which to reason for acting and planning. The former, called *end-to-end learning*, produces a reactive program that can be effective and efficient, and possibly amenable to continual adaptation; but it is usually a "black box" function, difficult to explain, verify or validate. The latter, in contrast, aims at acquiring explicit models that are predictive but not executable; they can support analysis and explanation.

For example, a robot collaborating with a human should be proved safe to its users. To be accepted as a co-worker, it should also be able to explain what it is doing and remain intelligible. End-to-end learning may be less adequate in that regard. However, it can be very useful for acquiring low-level reactive sensory-motor skills, e.g., for grasping and manipulation, with additional mechanisms for verification and validation. It can also be very useful for acquiring domain-dependent search heuristics for more efficient planning and acting.

1.1.6 Integrating Acting, Planning, and Learning

Acting, planning, and learning are connected in many different ways, seldom limited to a simple sequence $\langle learn, plan, act \rangle$. There is learning to plan and learning to act, but there is also acting to learn, and planning to learn. Let us mention a few possible interplays among these three functions.

An actor learns by acting. It may have the leisure to act for the sole purpose of learning. Possibly it may simulate its training actions to learn at an affordable cost. However, it is always desirable for an actor to keep learning while pursuing its activities, so it can improve and better adapt to a changing environment whose learned models need to be updated. Learning can be done when the actor fails, or when it can benefit from additional advice or knowledge.

An actor or its user may reason about better ways to learn—for example, by planning how to find states and activities that may be useful for learning. For example, *curriculum learning* targets a progressive and rationally organized learning program, or a well organized training database [112], as would be elaborated by an educator. Learning to learn, or *meta-learning* seeks to improve learning.

Often, an actor engaged in its tasks as well as in learning will have to find a tradeoff between learning more versus advancing in its task. This is the *exploration versus exploitation* tradeoff. An actor without much knowledge may favor exploration, while an expert actor may prefer to exploit known behaviors.

The *planning-to-learn* paradigm is important in this book. A learner can provide models and control knowledge, such as heuristics, to an online planning-acting duo.

[3]These are related to the notion *tacit knowledge*, e.g., how to recognize a face or ride a bicycle, as opposed to *explicit knowledge*, such as scientific facts and models [576].

Conversely, a planner can synthesize a number of random cases of problems and solutions to feed to a learner's training database. Planning can be used to create curricula for curriculum learning. In a continual-learning scheme, the actor's experiences are fed back to the original planner for use in additional training to improve what has been learned. These interactions, partially depicted in Figure 1.2, may possibly require different planners and interactions with a simulator as well as with the real world.

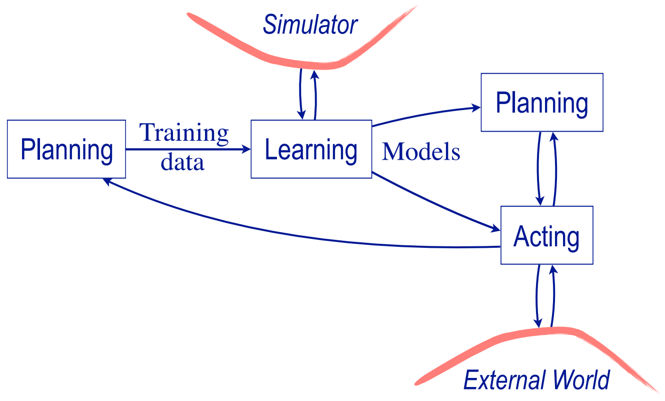


Figure 1.2. Interactions among acting, planning, and learning.

In some cases, a learner's output can be directly used for acting without additional planning. In these cases, the learner may synthesize from a training database a policy for reactive acting. This can be effective for focused and specialized functions, such as the sensory-motor control of a device. However, adaptation to a broad diversity of tasks and environments requires planning, hence it also requires learning for better planning, and possibly planning for learning as in the previous paragraph.

1.2 Descriptive and Operational Models of Actions

The book presents different models for acting, planning, and learning, starting from the simplest deterministic state-transition systems, to temporal, probabilistic and nondeterministic cases. The formal representations used for expressing these models will be introduced when needed. Most of the chapters use discrete models, except for Part VII which uses continuous models of motion and manipulation.

Actors' models of actions can be classified into two types:

- *Descriptive models* specify *what* effects an action may have and *when* it is feasible. Descriptive models, also called causal models, are relations from the precondition to the effects of an action. The actor uses these models during planning, to reason about what actions may achieve the actor's objectives.
- *Operational models* specify *how* to perform an action: what commands to execute in the current context, and how to organize them to achieve the action's intended effects. The actor uses these models during acting, to perform the actions that it has decided to perform.

Descriptive models are more abstract than operational models. They tend to ignore details, and focus on the main effects needed to decide about the eventual use of an action. For example, if you plan to take a book from a bookshelf, at planning time you usually are not concerned with the available space around the book to insert your fingers and extract the book. A descriptive model of an action abstracts away these details to focus on higher-level concerns, such as which shelf the book is in, whether it is within your reach, and whether you have a free hand with which to take it.

There are several reasons why these idealized abstract models are useful for planning. First, it is difficult to develop very detailed descriptive models. Second, these models may require information that is unknown during planning. Third, reasoning with detailed models is computationally very complex. Planners often need to search over many different combinations of actions, and if such a planner uses operational (rather than descriptive) models for this search, it may run very slowly.

Operational models of how to perform actions cannot do with the simplifications allowed in descriptive models. To pick up a book in a shelf, you will need to determine precisely where the book is located, whether you need to remove an obstacle to reach the book, which positions of your hand and fingers give a feasible grasp, and which sequences of precise motions and manipulations will allow you to perform the action.

Furthermore, operational models may need to include ways to respond to *exogenous* events, that is, events that occur because of external factors beyond the actor's control. For example, someone might be standing in front of the bookshelf, or the stool you intended to use to reach the book on a high shelf might be missing, or a potentially huge number of other possibilities might interfere with your plan.

In principle, descriptive models can take into account the uncertainty caused by exogenous events (see Parts III and IV). However, exogenous events are often ignored in descriptive models because it is impractical to try to model all of the possible joint effects of actions and exogenous events, or to plan in advance for all of the contingencies. In operational models, however, the need to handle exogenous events is much more compelling. Operational models must have ways to respond to such events if they happen, because they can interfere with the achievement of an action. In the bookshelf example, you might need to ask someone to move out of the way, or you might have to stand on a chair instead of the missing stool.

Finally, an actor's hierarchical organization and continual online processing can be integrated in these two types of models. We may have a hierarchy of operational models, sketching how to perform abstract tasks, and giving more detailed recipes for primitive actions. Similarly, we may have a hierarchy of descriptive models, from abstract tasks down to the effects of commands executable by the platform. Furthermore, deliberation may perform a continual and interleaved processing of operational models and descriptive models at different levels of the hierarchy. The book illustrates instances of these hierarchical models.

1.3 Responsible Research on Autonomous Actors

Autonomous deliberative actors are scientifically and technically challenging for AI. They are also ethically very challenging. We, and all contributors to AI, hold a particular responsibility regarding ethical issues. However, since no chapter of this book is devoted to ethics, we felt important to clarify here our position and concern, particularly regarding actor-centered AI.

Discussions of the ethics of AI are very active, with numerous publications, committees and recommendations (see for example [405, 176, 335, 1109]). Most of these discussions deal with data-centered ethical concerns, such as biases, privacy, fairness, transparency, trustworthiness, or ownership. They have been triggered by the significant AI advances in data interpretation and data generation. They are certainly very important. They need to be pursued and implemented into regulations (beyond the RGPD), institutions (e.g., data trusts) and active monitoring processes.

These data-centered ethical concerns are more focused on individuals than embracing broader social considerations, such as social cohesion, values, and democratic organization, which are becoming even more critical with the development of autonomous acting systems. Actor-related ethical issues may have more vital impacts on humanity—but they have not been as widely studied, possibly because of a less advanced state of the art.

Some of the actor-centered ethical issues are related to a possible automation of many human activities, including rewarding qualified professional and creative jobs. Such a trend, in particular if fast and widespread, would create economic problems about employment, inequalities and social wealth sharing. It would entail a questioning of our role in and value to society, and hence to ourself. Feeling socially superfluous, because machines might do most of what many people can do, may lead to significant human and social turmoil. It may cause infringements on human dignity.

Human interactions have already changed with social networks. They are fast changing with conversational agents becoming language-fluent and apparently knowledgeable. They will further change with the advent of autonomous actors that have not only the capabilities described earlier, but also have capable sensory-motor skills, detailed knowledge of a person, and can nudge or prod her with respect to dubious utility criteria. This prospect raises the risk of reduced autonomy and infringements on human freedom and agency.

Autonomous actors may possibly amplify inequalities and further tilt the power imbalance between human groups and nations. Leaders may be more likely to engage in conflicts if they can do so with no risks to their soldiers. Weaponized actors are a very serious concern. Despite a call from many scientists to ban lethal autonomous weapons [378], now supported by the UN and other organizations, there is unfortunately for the moment no international agreement on these matters. Strong opposition from most powerful nations remains.

Autonomous actors may also be beneficial to our well-being and health, for example as long-life empathic, serviceable and trustable companions. We need to remain proactively engaged towards these ends, but we must also keep in mind that the individual acceptance of a technology (even as a widespread market) is not equivalent to its social acceptance or acceptability. The latter must include, among other things, long-term effects, social cohesion and values, and environmental impacts.

Neither the best outcome nor the worst one are the most probable. However, our current social organization, and the profit-and-power motive for much of its development, do not lean naturally towards the best. To avoid the worst, we need to be well aware of the risks and be proactive in mitigating them.

A possible ambition is to seek machines aligned with human values [966, 232]. However, it is unclear whether it is feasible to have machines behaving with and enforcing our values, if their understanding of those values comes from our specifications or from observation of our inconsistent behaviors.[4] It is even more questionable whether we could put the risks of fast deployment on hold until we are able to have all of our AI machines human-aligned.

A more questionable option is to seek machines capable of moral choices. Machines do not have intrinsic motivations, desires, nor feelings with respect to which moral choices are meaningful. The so-called "ethics by design" can be quite misleading: techniques cannot solve everything, including our ethical choices and responsibilities. We certainly must improve and implement verification and validation methods towards provable trustworthiness, under appropriate assumptions. However, the responsibilities for designing, using, and allowing the deployment of AI actors remain ours. Researchers should not only be concerned with how AI should not be used for harmful purposes, but also with how it can be used to promote positive values and counteract antidemocratic and deceptive practices.

It is well known that technology is ambivalent, with both good and ugly faces.[5] Everyone in society is, to some degree, responsible for harmful technical deployments. Scientists hold particular responsibilities because they can investigate and foresee long term risks, and search for mitigating means. They can disseminate knowledge and be active in social debates about these risks. For that, we believe, they have to remain cautiously optimistic. This optimism is justified by the numerous expressions of risk-related concerns published by AI scientists and developers, and their calls for effective oversight and open independent verifications. It is also justified by some more advanced regulations (for example, the recently approved *European AI Act*). We recommend our responsible reader to remain actively vigilant.

[4] After centuries of moral effort, we have been able to state some of these values in documents such as the Universal Declaration of Human Rights. However, these rights are routinely violated and we are still unable to enforce them.

[5] Hephaestus, the Greek god of technology, is described as a limping deity.

Part I

Deterministic State-Transition Systems

. . . we must examine the nature of actions, namely how we ought to do them . . .

Aristotle, *Nichomachean Ethics*, circa 330 BCE

Any model of an actor and its environment is necessarily an imperfect approximation that must incorporate trade-offs among several competing criteria: accuracy, computational performance, and understandability to users. This part of the book uses a highly simplified model that incorporates the following set of restrictive assumptions:

1. *Finite, static world.* There are a finite set of possible states and a finite set of possible actions. The world changes from one state to another only when the actor performs an action. If the actor does not act, then the current state remains unchanged. This precludes the possibility of actions by other actors or exogenous events that are not due to any actor.
2. *Deterministic actions.* The outcome of performing an action a in a state s is determined solely by a and s, and the actor can predict this outcome with certainty. This excludes the possibility of accidents or execution errors, as well as nondeterministic actions, such as rolling a pair of dice.
3. *Full observability.* The actor always knows the current state of the world.[6]
4. *No explicit time, no concurrency.* There is no explicit model of time (e.g., when to start performing an action, how long a state or action should last, or how to perform other actions concurrently).[7] There is just a discrete sequence of states and actions.

[6]It has sometimes been argued that if the initial state is known, then determinism assures that the current state can always be inferred. That's nearly correct, but it also requires the actor to have perfect recall of the sequence of actions it has performed.

[7]This does not prohibit one from encoding time-related information (e.g., timestamps) into the states and the actions' preconditions and effects. However, to represent and reason about actions that have temporal durations, a more sophisticated representation is often needed (see Chapter 17).

Free pre-publication, for personal use only. To be published by Cambridge University Press.

These assumptions are used far more frequently in planning than in acting, and are often called the *classical planning assumptions*. An environment that satisfies them is called a *deterministic state-transition system* (for short, just a *state-transition system*) or a *classical planning domain*.

One consequence of these assumptions is that the actions in a state-transition system are necessarily more abstract than the actor's sensory-motor commands. A classical-planning model of an "open door" action may simply assert that the door is now open instead of closed, but a robot that opens a door will need to use sensory-motor commands that do not satisfy the classical-planning assumptions:

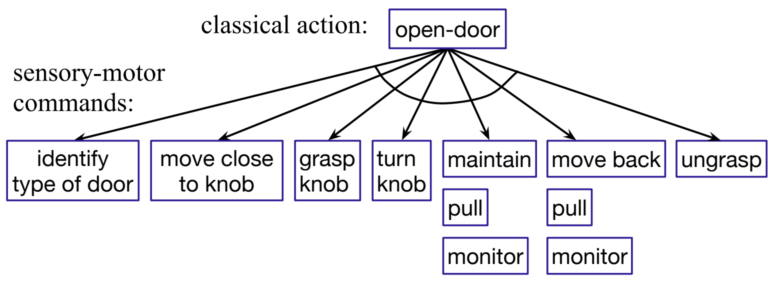


Figure I.1. A classical action, and non-classical commands that implement it.

Because few (if any) real-world environments satisfy all of the classical planning assumptions, deterministic state-transition models may introduce errors into an actor's deliberations. However, such models can still be desirable despite this problem. They usually are much simpler to construct and use than other kinds of domain models, and there are techniques for predicting some of the errors in advance so that an actor can change the plan to avoid them. Any remaining errors may be acceptable if they are infrequent and do not have severe consequences.

This part is organized as follows. Chapter 2 presents the *state-variable representation* that we will use, describes how it relates to several other representations, and then presents several acting algorithms. Chapter 3 presents and classifies several planning algorithms and related techniques, and Chapter 4 describes some ways to learn state-variable representations.

2 Deterministic Representation and Acting

This chapter is about representing state-transition systems and using them in acting. Section 2.2 gives formal definitions of state-transition systems and planning problems, and a simple acting algorithm. Section 2.3 describes state-variable representations of state-transition-systems, and Section 2.6 describes several acting procedures that use this representation. Section 2.4 describes classical representation, an alternative to state-variable representation that is often used in the planning literature.

2.1 Motivating Example

Most of the examples in this chapter will involve *Dock-Worker Robots (DWR)* domains. These are highly simplified "toy" versions of harbor and warehouse systems like the one in Figure 2.1. Depending on the example, the objects may include ships, cranes, loading docks, piles of containers, robot vehicles, roads, and delivery gates.

Example 2.1. Figure 2.2 shows a simple DWR domain that is a running example in this chapter. There are two robots, three loading docks, three containers, and three piles (stacks of containers). The domain has *states*, each of which is a configuration

Figure 2.1. A container terminal in Barcelona.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

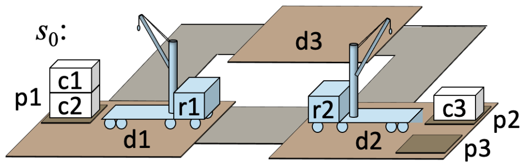


Figure 2.2. Running example: a simple DWR domain.

of the objects (as in the figure), and *actions* that cause transitions from one state to another. □

The next section is rather abstract, but we will return to this example in Section 2.3.

2.2 State-Transition Systems

Definition 2.2.[1] A *state-transition system*, or *classical planning domain*, is a tuple

$$\Sigma = (S, A, \gamma, \text{cost}) \quad \text{or} \quad \Sigma = (S, A, \gamma), \tag{2.1}$$

where

- S and A are the finite sets of *states* and *actions*,
- $\gamma : S \times A \to S$, the *state-transition function*, is a *partial function* (that is, its domain is a subset of $S \times A$) telling what state will be produced if the actor executes action a in state s. The set of *applicable* actions in s is[2]

$$\mathit{Applicable}(s) = \{a \in A \mid \gamma(s, a) \text{ is defined}\}. \tag{2.2}$$

- cost : $A \to [0, \infty)$, the *cost function*,[3] is a partial function having the same domain as γ. It may represent monetary cost, time, or some other numeric quantity that one might want to minimize. In the second form of Equation 2.1, in which the cost function is not given explicitly, $\text{cost}(a) = 1$ for every $a \in A$. □

2.2.1 Plans

In order to act purposefully, an actor will need some notion of what actions it needs to perform. In a deterministic state-transition model, this will be a *plan*: a finite sequence of actions

$$\pi = \langle a_1, \ldots, a_n \rangle. \tag{2.3}$$

The *length* of π is $|\pi| = n$, and the *cost* of π is the sum of the action costs:

$$\text{cost}(\pi) = \sum_{i=1}^{n} \text{cost}(a_i). \tag{2.4}$$

[1] Definitions, examples, and theorems are all numbered in the same numerical sequence. Algorithms and equations, however, are in separate sequences.

[2] Section 8.1 will change Equation 2.2 to model actions that have multiple possible outcomes.

[3] This definition prevents the cost from depending on s. See Remark 2.6 for a discussion of this restriction and some cases in which it can be lifted.

As a special case, $\langle\rangle$ is the *empty* plan, which contains no actions. Its length and cost are both 0.

A *subplan* π' of π is a (contiguous) subsequence[4] $\langle a_i, \dots, a_j\rangle$ of π. As special cases, the subplans $\langle a_1, \dots, a_i\rangle$ and $\langle a_j, \dots, a_n\rangle$ are a *prefix* and *suffix* of π, respectively.[5]

Here is some notation for concatenation of plans and actions. If $\pi = \langle a_1, \dots, a_n\rangle$ and $\pi' = \langle a'_1, \dots, a'_{n'}\rangle$ are plans and a is an action, then

$$\begin{aligned}
\pi \cdot a &= \langle a_1, \dots, a_n, a\rangle;\\
a \cdot \pi &= \langle a, a_1, \dots, a_n\rangle;\\
\pi \cdot \pi' &= \langle a_1, \dots, a_n, a'_1, \dots, a'_{n'}\rangle;\\
\pi \cdot \langle\rangle &= \langle\rangle \cdot \pi = \pi.
\end{aligned} \tag{2.5}$$

The state-transition function γ can easily be extended to include plans, by letting $\gamma(s, \pi)$ be the state produced by starting at s and applying the actions in π in the order that they are given, if all of them are applicable. More specifically:

- The empty plan $\langle\rangle$ is applicable in every state s, and $\gamma(s, \langle\rangle) = s$.
- If $\pi = a \cdot \pi'$, where a is applicable in s and π' is applicable in $\gamma(s, a)$, then π is applicable in s and

$$\gamma(s, \pi) = \gamma(\gamma(s, a), \pi'). \tag{2.6}$$

It immediately follows that if a plan $\pi = \langle a_1, a_2, \dots, a_n\rangle$ is applicable in a state s_0, then it produces a sequence of states $\langle s_0, s_1, \dots, s_n\rangle$ such that

$$s_1 = \gamma(s_0, a_1), \quad s_2 = \gamma(s_1, a_2), \quad \dots, \quad s_n = \gamma(s_{i-n}, a_n). \tag{2.7}$$

In this case, the *transitive closure* of π on s_0 is the path

$$\widehat{\gamma}(s_0, \pi) = \begin{cases} \langle s_0, s_1, \dots, s_n\rangle, & \text{if } \pi = \langle a_1, a_2, \dots, a_n\rangle,\\ \langle s_0\rangle, & \text{if } \pi = \langle\rangle. \end{cases} \tag{2.8}$$

From the usual definitions of the length and cost of a path, we get

$$|\widehat{\gamma}(s_0, \pi)| = |\pi|; \tag{2.9}$$

$$\text{cost}(\widehat{\gamma}(s_0, \pi)) = \text{cost}(\pi). \tag{2.10}$$

2.2.2 Planning Problems

A *classical planning problem* is a triple

$$P = (\Sigma, s_0, S_g), \tag{2.11}$$

[4]We use "subsequence" to mean a *contiguous* subsequence. This is consistent with our previous books [410, 411], but differs from the terminology in some other subfields of computer science [35, 258].

[5]This terminology is common in the AI planning literature, but it differs from ordinary English usage, in which a prefix or suffix would be something added to π, not part of π itself.

```
Run-Plan($\Sigma, \pi$)
    while True do
1       $s \leftarrow$ observe current state
        if $\pi = \langle\rangle$ then
2           return success
        $a \leftarrow$ pop($\pi$)
3       if $a \notin Applicable(s)$ then return failure
        perform action $a$
```

Algorithm 2.1. Run-Plan, a simple procedure to run a plan.

where Σ is a state-transition system, s_0 is a state called the *initial state*, and S_g is a set of states called *goal states*. A *solution* for P is any plan $\pi = \langle a_1, \dots, a_n \rangle$ such that $\gamma(s_0, \pi) \in S_g$. The solution π is *minimal* if no subsequence of π is also a solution, *shortest* if there is no solution π' such that $|\pi'| < |\pi|$, and *optimal* if

$$\text{cost}(\pi) = \min\{\text{cost}(\pi') \mid \pi' \text{ is a solution for } P\}. \tag{2.12}$$

Example 2.3. Suppose a planning problem P has three solution plans:

$$\pi_1 = \langle a_1 \rangle; \qquad \pi_2 = \langle a_2, a_3, a_4, a_5 \rangle; \qquad \pi_3 = \langle a_2, a_3, a_1 \rangle.$$

If each action's cost is 1, then π_1 is a minimal, shortest, and cost-optimal solution, π_2 is a minimal solution but is neither shortest nor cost-optimal, and π_3 is neither minimal nor shortest nor cost-optimal. □

2.2.3 Acting with a Plan

Algorithm 2.1, Run-Plan, is a simple procedure for running a plan. If π is applicable in the initial observed state,[6] then ideally it will produce the sequence of states $\widehat{\gamma}(s_0, \pi)$, and Run-Plan will return success. However, recall from Part I that Σ is not necessarily a perfect model of the actor's environment. Execution errors or unpredicted exogenous events may sometimes cause Run-Plan to encounter states in which the next action of π is not applicable, in which case Line 3 will return failure.

Run-Plan can be adapted to test whether π has achieved a desired goal S_g, by adding S_g to its argument list and adding the following line before Line 2:

if $s \notin S_g$ **then return** failure

Section 2.6 will discuss some ways for an actor to recover from failures, either by re-executing parts of π or acquiring a new solution plan. Chapter 3 will discuss several algorithms to produce solution plans.

[6] Although we call this the observed state, it is more likely to be an abstraction of the state that the actor observes, with various low-level details omitted that are irrelevant for planning.

2.3 State-Variable Representation

In Section 2.2, states were an abstract set $S = \{s_0, s_1, \ldots, \}$. Explicit enumeration of S can often be quite large; even trivially simple examples such as Figure 2.2 can have hundreds or thousands of states. Furthermore, names such as $s_0, s_1, \ldots$ tell us nothing about the states' internal structure.

To represent complex domains, we will want a more expressive representation that gives information about relationships among objects in the actor's environment. For example, to describe the state shown in Figure 2.2—which will be a running example in this section—we might want to write

$$\mathsf{loc}(\mathsf{r1}) = \mathsf{d1} \tag{2.13}$$

to mean that robot r1's location is d1.

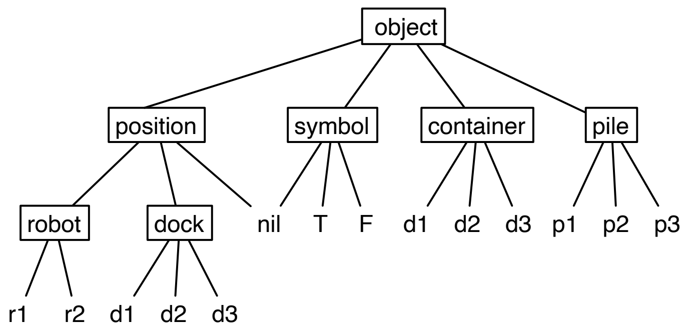


Figure 2.3. A type hierarchy for the objects in Figure 2.2. Boxed words are types, other words are objects.

In Equation 2.13, d1 and r1 are called *objects* or *object constants*. We will organize objects into sets and subsets using a type hierarchy. For example, Figure 2.3 is a type hierarchy for our running example, and it corresponds to the following sets of objects:

$$\mathcal{H} = \begin{cases} \mathit{Objects} = \mathit{Positions} \cup \mathit{Containers} \cup \mathit{Piles} \cup \mathit{Symbols}; \\ \mathit{Positions} = \mathit{Robots} \cup \mathit{Docks} \cup \{\mathsf{nil}\}; \\ \mathit{Symbols} = \{\mathsf{T}, \mathsf{F}, \mathsf{nil}\}; \qquad \mathit{Containers} = \{\mathsf{c1}, \mathsf{c2}, \mathsf{c3}\}; \\ \mathit{Piles} = \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}\}; \qquad \mathit{Robots} = \{\mathsf{r1}, \mathsf{r2}\}; \\ \mathit{Docks} = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}\}. \end{cases} \tag{2.14}$$

With a slight abuse in terminology, we not distinguish between types and and the corresponding sets of constants. For example, we will call $\mathcal{H}$ a type hierarchy.

We will have typed variables called *object variables*. In our running example, an object variable r of type robot has $\mathit{Range}(r) = \mathit{Robots}$ (or less formally, $r \in \mathit{Robots}$).

A domain usually has a set of *rigid* properties that do not change over time, such as its topology and connectivity. In our running example, robots and containers can be moved around, but the locations of piles and docks are rigid. To represent such properties we will define *rigid relations* over the types. In our running example, the rigid relations are $\mathsf{adjacent} \subseteq \mathit{Docks} \times \mathit{Docks}$, a symmetric relation telling which pairs of loading docks have roads between them, and $\mathsf{at} \subseteq \mathit{Piles} \times \mathit{Docks}$, which gives each pile's location:

$$\begin{aligned}\mathsf{adjacent} &= \{(\mathsf{d1},\mathsf{d2}),(\mathsf{d2},\mathsf{d1}),(\mathsf{d2},\mathsf{d3}),(\mathsf{d3},\mathsf{d2}),(\mathsf{d3},\mathsf{d1}),(\mathsf{d1},\mathsf{d3})\};\\ \mathsf{at} &= \{(\mathsf{p1},\mathsf{d1}),(\mathsf{p2},\mathsf{d2}),(\mathsf{p3},\mathsf{d2})\}.\end{aligned}$$

Properties of the domain that change over time, possibly under the effect of the actor's activity, we will describe using functional terms called *state variables*.[7] State variables have zero or more arguments, each of which may be either an object or an object variable. Each state variables is typed; for example, $\mathsf{loc}(\mathsf{r1}) \in \mathit{Docks}$.

2.3.1 Representing States

In each state, every ground state variable has a value that we will represent as an *assignment*, written as an equation similar to Equation 2.13, which assigns to $\mathsf{loc}(\mathsf{r1})$ the value $\mathsf{d1}$. We also will find it useful to refer to *lifted assignments*, e.g., $\mathsf{loc}(\mathsf{r1}) = l$, where r and l are object variables to be instantiated later (see Definition 2.5).

Here are the state variables. Their parameter types are $r \in \mathit{Robots}$, $d \in \mathit{Docks}$, $c \in \mathit{Containers}$, $p \in \mathit{Piles}$.

- $\mathsf{cargo}(r) \in \mathit{Containers} \cup \{\mathsf{nil}\}$ is either the container that r is carrying, or nil if r is not carrying anything. Each robot can hold at most one container.
- $\mathsf{loc}(r) \in \mathit{Docks}$ is the loading dock where r is located.
- $\mathsf{occupied}(d) \in \{\mathsf{T},\mathsf{F}\}$ is T if there is a robot at d, and F otherwise. At most one robot can be at each loading dock.
- $\mathsf{pile}(c) \in \mathit{Piles} \cup \{\mathsf{nil}\}$ is the pile that c is in, or nil if c is not in a pile.
- $\mathsf{pos}(c) \in \mathit{Robots} \cup \mathit{Piles} \cup \{\mathsf{nil}\}$ is c's position, which may be a robot, another container if c is in a pile, or nil if c is at the bottom of a pile.
- $\mathsf{top}(p) \in \mathit{Containers} \cup \{\mathsf{nil}\}$ is the container at the top of p, with $\mathsf{top}(p) = \mathsf{nil}$ if p is empty.

Example 2.4. The following total assignment is the state shown in Figure 2.2:

$$\begin{aligned}s_0 = \{&\mathsf{cargo}(\mathsf{r1})=\mathsf{nil}, && \mathsf{cargo}(\mathsf{r2})=\mathsf{nil}, &&\\ &\mathsf{loc}(\mathsf{r1})=\mathsf{d1}, && \mathsf{loc}(\mathsf{r2})=\mathsf{d2}, &&\\ &\mathsf{occupied}(\mathsf{d1})=\mathsf{T}, && \mathsf{occupied}(\mathsf{d2})=\mathsf{T}, && \mathsf{occupied}(\mathsf{d3})=\mathsf{F},\\ &\mathsf{pile}(\mathsf{c1})=\mathsf{p1}, && \mathsf{pile}(\mathsf{c2})=\mathsf{p1}, && \mathsf{pile}(\mathsf{c3})=\mathsf{p2},\\ &\mathsf{pos}(\mathsf{c1})=\mathsf{c2}, && \mathsf{pos}(\mathsf{c2})=\mathsf{nil}, && \mathsf{pos}(\mathsf{c3})=\mathsf{nil},\\ &\mathsf{top}(\mathsf{p1})=\mathsf{c1}, && \mathsf{top}(\mathsf{p2})=\mathsf{c3}, && \mathsf{top}(\mathsf{p3})=\mathsf{nil}\}.\end{aligned} \quad \square$$

Usually some total assignments are nonsensical in the domain that the state variables are intended to represent. In our running example, it is nonsensical to have a state in which both $\mathsf{pos}(\mathsf{c1}) = \mathsf{r1}$ and $\mathsf{cargo}(\mathsf{r1}) = \mathsf{F}$. In principle, one could exclude such

[7] The terms *state variable* and *fluent* are considered synonymous [967]. However, in much of the published literature, fluents have only Boolean values, and state variables have no parameters. We use a more flexible representation that includes both parameters and non-Boolean values.

things from the set of states S by writing a set of constraints that every state must satisfy. However, we will instead will leave these “unreal” states in S, and enforce the constraints implicitly by writing actions that always map real states to other real states (see Section 2.3.3).

Definition 2.5. The following terminology is borrowed loosely from first-order logic:

- An *atom* (short for *atomic formula*) or *positive literal* is either a rigid-relation assertion $rel(z_1, \ldots, z_n)$, or a state-variable assignment $x(z_1, \ldots, z_{n-1}) = z_n$, where *rel* or x is the name of the relation or state variable, and $z_1, \ldots, z_n$ are objects or object variables.
- A *negated* atom or *negative literal* is an atom with a negation sign in front of it, such as $\neg rel(z_1, \ldots, z_n)$ or $\neg\, x(z_1, \ldots, z_{n-1}) = z_n$. We usually will write the latter as $x(z_1, \ldots, z_{n-1}) \neq z_n$.
- Let e be any syntactic expression that contains literals. Then e is *ground* if it contains no object variables, *lifted* if it contains object variables but no objects, and *partially instantiated* if it includes both objects and object variables.
- If z is an object variable in e, then *instantiating* z means replacing z with either an object in $Range(z)$, or another object variable z' such that $Range(z') \subseteq Range(z)$. Instantiating e means instantiating zero or more of the object variables in e. The resulting expression is an *instance* of e. □

Remark 2.6. Although state variables and rigid relations may have arguments that are object constants or object variables, we do not—for now—allow the arguments to be other state variables, as in an expression such as $\mathsf{at}(\mathsf{p1}, \mathsf{loc}(\mathsf{r1}))$.[8] This restriction, and the restriction in Definition 2.2 that the cost function cannot depend on s, are needed to accommodate the requirements of some, but not all, of the algorithms in Parts I, II, and VI. Cases in which these restrictions can be relaxed or discarded are discussed in Sections 2.7.2 and 3.6.7 and the first paragraph of Chapter 5. □

2.3.2 Action Schemas and Actions

Definition 2.7. Given a type hierarchy $\mathcal{H}$, an *action schema* (or *action template*) is a tuple

$$\alpha = (\mathrm{head}(\alpha), \mathrm{pre}(\alpha), \mathrm{eff}(\alpha), \mathrm{cost}(\alpha)) \quad \text{or} \quad \alpha = (\mathrm{head}(\alpha), \mathrm{pre}(\alpha), \mathrm{eff}(\alpha)),$$

where:

- $\mathrm{head}(\alpha)$ is an expression of the form $\mathit{name}(z_1, \ldots, z_k)$, where *name* is a name and $(z_1, \ldots, z_k)$ is a list of zero or more object variables that are α’s *parameters*. So that *name* will uniquely identify α, no other action can have the same name.
- $\mathrm{pre}(\alpha) = \{p_1, \ldots, p_m\}$ is a set of zero or more *preconditions*, each of which is a literal. Within each literal p_i, every argument must be either an object or one of the parameters $z_1, \ldots, z_k$.

[8]Were it not for this restriction, the definition of instantiation in Definition 2.5 would also allow substituting a state variable x for an object variable z if $Range(x) \subseteq Range(z)$.

- eff(α) = $\{x_1 = v_1, \ldots, x_n = v_n\}$ is a set of zero or more *effects*, each of which is a state-variable assignment for a different state variable. In each effect, the assigned value v_i must be either an object or one of $z_1, \ldots, z_k$.
- cost(α) is a positive number denoting the cost of applying actions that are instances of α. If it is omitted from the schema, it defaults to 1.
- Every parameter and state variable in α has a range that is one of the sets in $\mathcal{H}$.

Notation and terminology. To emphasize that each effect $x_i = v_i$ changes a state variable's value, we usually will instead write it as $x_i \leftarrow v_i$. Furthermore, we usually will write α in the following pseudocode format, omitting the last line if $c = 1$:

```
$\mathit{name}(z_1, z_2, \ldots, z_k)$
   pre: $p_1, \ldots, p_m$
    eff: $x_1 \leftarrow v_1, \ldots, x_n \leftarrow v_n$
  cost: $c$
```

We will often refer to α by writing just its name, and to instances of α by writing just their heads, as in the following two examples. Such references are unambiguous because α's name is unique and its only variables are its parameters. □

Example 2.8. Continuing Example 2.1, here are three action schemas, where $c \in$ *Containers*, $c' \in$ *Containers* $\cup$ {nil}, $d, d' \in$ *Docks*, $p \in$ *Piles*, and $r \in$ *Robots*:

```
take$(r, c, c', p, d)$                                  // $r$ takes $c$ off of $p$
   pre: at$(p, d)$, cargo$(r)$ = nil, loc$(r) = d$, pos$(c) = c'$, top$(p) = c$
    eff: cargo$(r) \leftarrow c$, pile$(c) \leftarrow$ nil, pos$(c) \leftarrow r$, top$(p) \leftarrow c'$

put$(r, c, c', p, d)$                                   // $r$ puts $c$ onto $p$
   pre: at$(p, d)$, pos$(c) = r$, loc$(r) = d$, top$(p) = c'$
    eff: cargo$(r) \leftarrow$ nil, pile$(c) \leftarrow p$, pos$(c) \leftarrow c'$, top$(p) \leftarrow c$

move$(r, d, d')$                                        // $r$ moves from $d$ to $d'$
   pre: adjacent$(d, d')$, loc$(r) = d$, occupied$(d')$ = F
    eff: loc$(r) \leftarrow d'$, occupied$(d) \leftarrow$ F, occupied$(d') \leftarrow$ T
```

In take and put, one might be tempted to replace the preconditions at(p, d) and loc$(r) = d$ with a single precondition at$(p, \text{loc}(r))$. In other situations, one might want to put computational formulas in the preconditions and effects of action schemas. The restriction in Remark 2.6 prevents these things, but later in the book we will discuss cases where the restriction can be relaxed. □

Let a be *ground instance* of an action schema, that is, a is an expression produced by substituting object variables for all of the action schema's parameters. If a is a ground instance of an action schema and eff(a) assigns at most one value to each state variable, then a represents an action. If a state s satisfies pre(a), then a is *applicable* in s, and applying it produces the following state:

$\gamma(s, a)$ = {an assignment $x = w$ for each effect $x \leftarrow w$ in eff(a)} $\cup$ {every assignment $x = w$ in s such that $\text{eff}(a)$ does not assign a value to $x\}$. (2.15)

Thus for every assignment $x = v$ in s,

$$\gamma(s,a) \text{ contains } \begin{cases} x = w & \text{if eff}(a) \text{ contains } x \leftarrow w \text{ for some } w, \\ x = v & \text{otherwise.} \end{cases}$$

If a isn't applicable in s, then $\gamma(s,a)$ is undefined.

Example 2.9. Let $\mathcal{A}$ be the set of action schemas in Example 2.8, and let a_1 = take(r1, c1, c2, p1, d1). Then

$$\begin{aligned} \text{pre}(a_1) &= \{\text{at}(\text{p1},\text{d1}), \text{cargo}(\text{r1}) = \text{nil}, \text{loc}(\text{r1}) = \text{d1}, \text{pos}(\text{c1}) = \text{c2}, \text{top}(\text{p1}) = \text{c1}\}; \\ \text{eff}(a_1) &= \{\text{cargo}(\text{r1}) \leftarrow \text{c1}, \text{pile}(\text{c1}) \leftarrow \text{nil}, \text{pos}(\text{c1}) \leftarrow \text{r1}, \text{top}(\text{p1}) \leftarrow \text{c2}\}. \end{aligned}$$

It follows that a_1 is applicable to the state s_0 in Example 2.4. The state $\gamma(s_0, a_1)$ is shown in Figure 2.4. It is identical to s_0 except for the following changes:

$$\text{cargo}(\text{r1}) = \text{c1},\ \text{pile}(\text{c1}) = \text{nil},\ \text{pos}(\text{c1}) = \text{r1},\ \text{top}(\text{p1}) = \text{c2}. \qquad \square$$

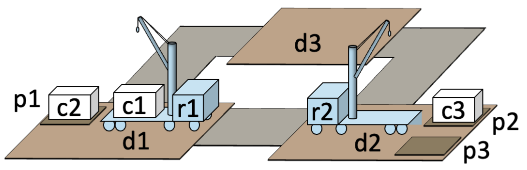


Figure 2.4. The state $\gamma(s_0, a_1)$, where s_0 and a_1 are as in Example 2.9.

2.3.3 Representing Planning Problems

In Section 2.2 we defined planning problems using a set of goal states S_g. To represent S_g we will use a set of ground literals g called a *goal formula*, with S_g being the set of all states that satisfy g, that is, $S_g = \{s \in S \mid s \models g\}$.[9]

For notational convenience, we will usually write a call to a planning algorithm as

$$\mathit{Planner}(\Sigma, s_0, g) \quad \text{or} \quad \mathit{Planner}(\Sigma, s_0, S_g),$$

where *Planner* is the name of the planning algorithm and (Σ, s_0, g) or (Σ, s_0, S_g) is the planning problem. However, as we explained at the start of Section 2.3, what the planner needs is not an exhaustive list of everything in Σ, but instead a compact representation with which it can quickly compute the parts of Σ that it needs. In most cases, the following information is sufficient:

- a type hierarchy $\mathcal{H}$,
- a set R of rigid relations,

[9]Obviously this places some limitations on what states can be in S_g. A widely used work-around is to add state variables to the domain to make it easier to represent important sets of states. An example is the cargo state variable in Example 2.8.

- a set X of state variables, including specifications of their ranges,
- a set $\mathcal{A}$ of action schemas,
- an initial state s_0,
- a goal formula g.

More specifically:

Definition 2.10. $(\mathcal{H}, R, X, \mathcal{A})$ is a *state-variable representation* of a state-transition system $\Sigma = (S, A, \gamma, \text{cost})$ in which S contains all total assignments of the state variables, A is the set of all actions represented by the action schemas in $\mathcal{A}$, and γ is given by Equation 2.15. Similarly, $(\mathcal{H}, R, X, \mathcal{A}, s_0, g)$ is a state-variable representation of the planning problem $P = (\Sigma, s_0, g)$.

Terminology. When using a state-variable representation for Σ, we will sometimes call Σ a *state-variable planning domain*. In this case, Σ is *lifted* if both X and $\mathcal{A}$ are lifted, and *ground* if both X and $\mathcal{A}$ are ground. □

In a state-variable representation, some of the total assignments in S may be nonsensical. For example, let $(\mathcal{H}, R, X, \mathcal{A})$ be the state-variable representation developed in Examples 2.1 and 2.8. In the environment that $(\mathcal{H}, R, X, \mathcal{A})$ is intended to represent, it would be nonsensical to have a state in which both $\mathsf{cargo(r1) = c1}$ and $\mathsf{loc(c1) = d3}$. The representation allows such states, but none of them can ever be reached from states such as the one in Figure 2.2.

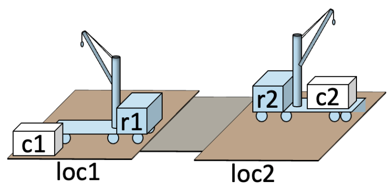
In principle, one could formulate constraints to exclude nonsensical states from S. However, if the initial state s_0 and the action schemas in $\mathcal{A}$ are defined properly, then no plan that begins at s_0 will ever produce a nonsensical state. Thus such constraints generally are unnecessary. In the AI planning literature, most of the classical-planning formulations do not use constraints on states.

2.4 Classical Representation

Classical representation[10] is an alternative to state-variable representation that has been widely used in the literature on automated planning. It differs from state-variable representation primarily in the following respects. All atoms have a name-and-arguments syntax, with no '=' or '←' symbols. Each state s is represented as the set of all atoms that are true in s, hence any atom not in this set is false in s. Each *planning operator* (the classical version of an action schema) has preconditions and effects that contain atoms and negated atoms.

Figure 2.5 shows state-variable and classical representations of a simple DWR planning problem. In pedagogical examples like the one in the figure, there usually are no type declarations in the classical-planning domain, depending instead on the actions to produce sensible values for the variables. However, computer implementations of these examples usually do include type declarations (see Section 2.4.1).

[10]This is also sometimes called "STRIPS representation" because it is similar (though not identical) to the representation used in the STRIPS planning system [359, 856].



(a) Initial state s_0

Types:

$\mathit{Robots} = \{\mathsf{r1},\mathsf{r2}\}$, $\mathit{Containers} = \{\mathsf{c1},\mathsf{c2}\}$,
$\mathit{Locs} = \{\mathsf{loc1},\mathsf{loc2}\}$, $\mathit{Booleans} = \{\mathsf{T},\mathsf{F}\}$

$r \in \mathit{Robots}$; $c \in \mathit{Containers}$; $l, m \in \mathit{Locs}$;
$\mathsf{loc}(r) \in \mathit{Locs}$;
$\mathsf{pos}(c) \in \mathit{Locs} \cup \mathit{Robots}$;
$\mathsf{loaded}(r) \in \mathit{Booleans}$

$\mathsf{take}(r,c,l)$
pre: $\mathsf{loc}(r,l)$, $\mathsf{pos}(c,l)$, $\neg\mathsf{loaded}(r)$
eff: $\mathsf{pos}(c,l)$, $\neg\mathsf{pos}(c,r)$, $\mathsf{loaded}(r)$

$\mathsf{put}(r,c,l)$
pre: $\mathsf{loc}(r,l)$, $\mathsf{pos}(c,r)$
eff: $\mathsf{pos}(c,l)$, $\neg\mathsf{pos}(c,r)$, $\neg\mathsf{loaded}(r)$

$\mathsf{move}(r,l,m)$
pre: $\mathsf{loc}(r,l)$
eff: $\mathsf{loc}(r,m)$, $\neg\mathsf{loc}(r,l)$

$$s_0 = \{\mathsf{loc(r1,loc1)}, \mathsf{loc(r2,loc2)}, \mathsf{pos(c1,loc1)}, \mathsf{pos(c2,r2)}, \mathsf{loaded(r2)}\}$$

$$g = \{\mathsf{pos(c1,loc2)}\}$$

(b) Classical representation

$\mathsf{take}(r,c,l)$
pre: $\mathsf{loc}(r) = l$, $\mathsf{pos}(c) = l$, $\mathsf{loaded}(r) = \mathsf{F}$
eff: $\mathsf{pos}(c) \leftarrow r$, $\mathsf{loaded}(r) \leftarrow \mathsf{T}$

$\mathsf{put}(r,c,l)$
pre: $\mathsf{loc}(r) = l$, $\mathsf{pos}(c) = r$
eff: $\mathsf{pos}(c) \leftarrow l$, $\mathsf{loaded}(r) \leftarrow \mathsf{F}$

$\mathsf{move}(r,l,m)$
pre: $\mathsf{loc}(r) = l$
eff: $\mathsf{loc}(r) \leftarrow m$

$$s_0 = \{\mathsf{loc(r1)} = \mathsf{loc1}, \mathsf{loc(r2)} = \mathsf{loc2}, \mathsf{pos(c1)} = \mathsf{loc1}, \mathsf{pos(c2)} = \mathsf{r2}, \mathsf{loaded(r1)} = \mathsf{F}, \mathsf{loaded(r2)} = \mathsf{T}\}$$

$$g = \{\mathsf{pos(c1)} = \mathsf{loc2}\}$$

(c) State-variable representation

Figure 2.5. State-variable and classical representations of a simple planning problem. It is similar to Example 2.8, but with the following differences: there are no piles, containers cannot be stacked on each other, and both robots may be at the same location.

Instead of a list of positive and negative effects, classical planning operators sometimes are written with lists of atoms to add and delete from the current state. For example, in the take operator in Figure 2.5(b), the 'eff:' line may be replaced with

add: $\mathsf{loaded}(r)$, $\mathsf{pos}(c,l)$
del: $\mathsf{pos}(c,r)$

Classical and state-variable representations have equivalent expressive power. Each can be translated into the other with at most a polynomial increase in the size of the representation. Because of this, the computational complexity results in Section 2.5 are independent of whether the planning problems are represented in classical or state-variable representation.

2.4.1 PDDL Example

PDDL, the Planning Domain Definition Language, is based on classical representation but uses a LISP-like syntax. As an example, Figure 2.6 shows a PDDL version of the planning problem in Figure 2.5. The purpose of the `requirements` clause at the beginning of the domain definition is to specify what capabilities a planning system

```
(define (domain example-domain)

  (requirements
     :negative-preconditions)
  (:action take
     :parameters (?r ?l ?c)
     :precondition
         (and (loc ?r ?l)
              (loc ?c ?l)
              (not (loaded ?r)))
     :effect
         (and (not (loc ?c ?l))
              (loc ?c ?r)
              (loaded ?r)))
  (:action put
     :parameters (?r ?l ?c)
     :precondition
         (and (loc ?r ?l)
              (loc ?c ?r))
     :effect
         (and (loc ?c ?l)
              (not (loc ?c ?r))
              (not (loaded ?r))))
```

```
  (:action move
     :parameters (?r ?l ?m)
     :precondition
         (and (loc ?r ?l)
              (adjacent ?l ?m))
     :effect
         (and (not (loc ?r ?l))
              (loc ?r ?m))))


(define (problem example-prob)
  (:domain example-domain))
  (:init
      (adjacent loc1 loc2)
      (adjacent loc2 loc1)
      (loc c1 loc1)
      (loc c2 r2)
      (loc r1 loc1)
      (loc r2 loc2))
  (:goal (loc c1 loc2)))
```

Figure 2.6. PDDL representation of the classical planning problem in Figure 2.5.

Figure 2.7. Complexity of classical planning problems.

Is P lifted or ground?	Is Σ fixed in advance?	Complexity of PLAN EXISTENCE	Complexity of PLAN LENGTH
lifted	no	EXPSPACE-complete	NEXPTIME-complete
ground	no	PSPACE-complete	PSPACE-complete
lifted	yes	PSPACE	PSPACE
ground	yes	Constant time	Constant time

will need. Here, it specifies that the planner must be able to reason about negative preconditions such as `(not (loaded ?r))` in the `take` operator.

The domain definition in Figure 2.6 does not include a type hierarchy like the one in Figure 2.5(b). However, PDDL provides an option for specifying one, by including `:typing` in the `requirements` clause. PDDL also includes ways to write axioms for inferring properties that are not stated explicitly, preferences on which goals to achieve, elementary numeric operations, certain kinds of temporal constraints and deterministic exogenous events. For tutorial expositions of these and other features, see [481].

2.5 Computational Complexity

Computational complexity results are normally given for *decision problems*, where each decision problem is an infinite set questions that have yes/no answers. Here two decision problems in which P may be any state-variable planning problem:

- PLAN EXISTENCE: does P have a solution?
- PLAN LENGTH: does P have a solution of length $\leq k$?

Figure 2.7 shows how the computational complexity of each decision problem P depends on whether the problem is lifted or ground, and whether the planning domain is given in the planner's input or fixed in advance (thus allowing a domain-specific planner to be used). Section 2.7.1 provides additional information.

The lower computational complexity values when P is ground do not mean that grounding a decision problem will make it easier to solve. Computational complexity is relative to the size of the problem representation, which is much larger for the grounded version of a problem than the lifted version, so grounding a problem makes makes its computational complexity smaller even though the amount of computation to solve it remains roughly the same.

Although these complexity results may look intimidating, they are *worst-case* results. There are many planning domains in which the time complexity is much lower (for example, many are polynomial in the average case, and some are polynomial even in the worst case). Furthermore, there are planning algorithms (such as variations of GBFS in Section 3.1.6) that can often find near-optimal solutions very quickly.

Because of those considerations, it might be more useful to think of the complexity results as indications of the expressivity of state-variable representation. Despite all of its restrictions, it is capable of expressing problems that are very hard to solve.

2.6 Acting

Suppose an actor calls a planning algorithm on a planning problem $P = (\Sigma, s_0, S_g)$, and the planner returns a solution plan π. If Σ were a perfect model of the environment, π would be guaranteed to produce the state $\gamma(s_0, \pi)$. However, because it is very unlikely that Σ will model the environment perfectly, unexpected outcomes might occur. These may be caused, for example, by problems with the actor's execution platform, incorrect information in the actor's model of the world, or exogenous events. In such situations, an actor can sometimes react by selectively choosing which parts of π to execute, as described in the next section. In other cases, the actor may need to call a planner to get a new plan, as discussed in Section 2.6.2.

2.6.1 Reactive Plan Execution

Sometimes an actor can react to unexpected events during plan execution by repeatedly choosing which parts of π to execute—for example, by re-executing some actions or skipping some actions. Algorithm 2.2, Reactive-Execution, is a procedure to do this. In the **for** loop at Line 2, it searches for a suffix $\langle a_i, \ldots, a_n \rangle$ of π that can achieve g from the current state s. It returns failure if the search is unsuccessful, and otherwise it executes a_i, gets the new current state, and repeats the search. The **for** loop at Line 2 is inefficient because it recomputes many of the same state transitions at each iteration of the loop, but Exercise 2.5 looks at some ways to speed it up.

```
Reactive-Execution($\Sigma, \pi, g$)
    let $\langle a_1, \ldots, a_n \rangle$ be the actions in $\pi$
1   while True do
        $s \leftarrow$ observe current state
        if $s \models g$ then return success
        $a \leftarrow$ nil
2       for $i \leftarrow n$ down to 1 do
            if $\gamma(s, \langle a_i, \ldots, a_n \rangle) \models g$ then
                $a \leftarrow a_i$
                exit the for loop
        if $a$ = nil then return failure
        perform $a$
```

Algorithm 2.2. Reactive-Execution is an acting procedure that selects and executes parts of a plan π, repeating until it either achieves a goal g or fails.

Reactive-Execution can react very quickly in situations where parts of π are still capable of achieving the goal. In other situations, it may be necessary for the actor to acquire a new or modified plan. We now will discuss some acting procedures that do this by calling a planner that can be used online.

2.6.2 Acting with Lookahead

```
Run-Lookahead($\Sigma, g$)
    while True do
        $s \leftarrow$ observed current state
        if $s \models g$ then return success
        $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
        if $\pi$ = failure then return failure
        $a \leftarrow$ pop-first-action($\pi$)    // remove and return $\pi$'s first action
        trigger the execution of $a$
```

Algorithm 2.3. Run-Lookahead, which replans before each action.

This section discusses two procedures based on the receding-horizon approach described in Section 1.1.4. Both procedures use an online planning algorithm, *Lookahead*, that is not required to return an entire solution plan. The plan may go part of the way to a goal state, or may even be a single action. Section 2.6.3 will discuss some ways to modify the planning algorithms in Chapter 3 to do this.

The first procedure is Algorithm 2.3, Run-Lookahead. Until it reaches a goal state, it repeatedly calls *Lookahead* to get a plan, performs the first action of the plan, and calls *Lookahead* again. This can be useful if the environment often changes unpredictably in ways that can cause plans to fail, because it immediately detects such changes and replans. However, it might be impractical if *Lookahead* has a large running time, and it might be unnecessary if plan failures are infrequent.

```
Run-Lazy-Lookahead($\Sigma, g$)
    $\pi \leftarrow \langle\rangle$
    while True do
        $s \leftarrow$ observed state
        if $s \models g$ then return success
1       if $\pi = \langle\rangle$ or Simulate($\Sigma, s, g, \pi$) = failure then
2           $\pi \leftarrow$ Lookahead($\Sigma, s, g$)
            if $\pi$ = failure then return failure
        $a \leftarrow$ pop-first-action($\pi$)      // remove and return $\pi$'s first action
        perform $a$
```

Algorithm 2.4. Run-Lazy-Lookahead, which replans only when necessary.

The second procedure is Algorithm 2.3, Run-Lazy-Lookahead. Repeatedly, it gets a plan π from *Lookahead* and executes π until either it reaches a goal and exits, or π ends, or *Simulate*, a plan simulator, says the rest of π will not work properly.

The purpose of *Simulate* is to detect potential future problems before they occur. A simple *Simulate* program could return success if $\gamma(s, \pi) \models g$ and failure otherwise. To detect more subtle problems, *Simulate* could instead do a more detailed test such as a physics-based simulation.

Compared to Run-Lookahead, Run-Lazy-Lookahead eliminates the overhead of planning at every step and replaces it with the overhead of running *Simulate* at every step. If *Simulate* is not too complicated, this will probably reduce the total overhead since it evaluates a single plan rather than a large space of plans. However, a potential advantage of Run-Lookahead is that it can respond to exogenous changes that make a better plan available, such as an unexpected opportunity to take a faster route to a destination or get a higher score in a game.

Both Run-Lookahead and Run-Lazy-Lookahead interleave acting and planning. It is also possible to write procedures in which the acting and planning processes run concurrently. This is more complicated because of the need to coordinate the two processes, but it can be useful in rapidly changing environments.

2.6.3 Interacting with an Online Planner

We emphasized earlier that *Lookahead* should be an online planner: the actor may call it frequently to get updated plans, and it may need to produce plans quickly so that the actor doesn't have to make long pauses between actions. However, most of the planning algorithms to be discussed in Chapter 3 are designed to run offline: they return only when they have found a solution plan or verified that no solution exists. In rapidly changing environments, the soundness of such planners can no longer be guaranteed: by the time that the planner returns a plan, changes to the environment may already have invalidated it. To use such a planner online, one may want to modify the planner or how the actor interacts with it. We now discuss some possible modifications.

Subgoaling. One way that the actor can reduce the amount of time used by the planner is to call it on smaller planning problems. Instead of giving a planner the ultimate goal g, the actor may instead give the planner a *subgoal* that needs to be achieved in order to achieve g. Once the subgoal has been achieved, the actor may formulate its next subgoal and call the planner again.

In practical settings, formulating these subgoals is usually done in a domain-specific manner. However, one possible domain-independent approach may be to compute an ordered set of landmarks (see Section 3.2.3) and choose the earliest one as a subgoal.

Limited-horizon planning. Recall that in the receding-horizon technique, the planner starts at the current state and searches until it either reaches a goal or exceeds the planning horizon, then it returns the best solution or partial solution it has found. Several of the planning algorithms in Chapter 3 can be modified to do this.

The term *partial solution* is somewhat misleading because there is no guarantee that the plan will actually lead the actor toward a goal. However, even a complete solution plan will not always enable an actor to reach the goal, because the actor may encounter problems that are not in the planner's domain model.

Sampling. In a *sampling* search, the planner uses a modified version of greedy search (Section 3.1.2) in which the node selection is randomized. The choice can be purely random, or it can be weighted according to a heuristic evaluation (see Chapter 3). The modified algorithm can do this several times to generate multiple solutions, and either return the one that looks best or return the n best solutions so that the actor can evaluate them further. The UCT and UPOM algorithms in Chapters 9 and 15 use this technique.

2.6.4 Acting with Plan Repair

When an actor runs into problems while executing a plan π, sometimes it is preferable to repair π instead of calling *Lookahead* to get a new one. This can reduce the runtime needed for planning, and can also improve the plan's *stability*, that is, the amount of the original plan π that is retained in the repaired plan. While executing π, the actor might have made commitments to other actors that would be difficult to cancel, or may have obtained resources that are needed later in π and are important not to waste. Plan stability can also be important for human interaction, as users may be confused if an actor makes radical changes to π in response to trivial problems.

To modify Run-Lazy-Lookahead to try to repair plans, Line 2 can be replaced with

$$\pi \leftarrow \textit{Lookahead-Repair}(\Sigma, s, g, \pi)$$

where *Lookahead-Repair* should attempt to repair π, and return a new plan only if its repair attempts fail. Section 3.5 discusses some possible plan-repair algorithms.

2.7 Discussion and Bibliographic Notes

2.7.1 Classical and State-Variable Representations

Although problem representations based on state variables have long been used in control-system design [440, 992, 305] and operations research [1075, 7, 519], their use in automated-planning research came much later [69, 71, 391]. Instead, most automated-planning research has used representation and reasoning techniques derived from mathematical logic. This began with the early work on GPS [843] and the situation calculus [771] and continued with the STRIPS planning system [359] and the classical representation described in Section 2.4 [856, 883, 715, 410, 967]. Classical representation is sometimes called STRIPS representation, but it is somewhat simpler than the representation used in the STRIPS planner [359].

In classical and state-variable planning domains, it is possible to encode (rather awkwardly) arithmetic relations among finite sets of numbers [481]. State-variable representations can be extended to include real numeric state variables, but this incurs a sharp increase in computational complexity [489].

The PDDL representation language was first published in 1998 [409] for use in the AIPS-98 planning competition [735], the first of a long series of International Planning Competitions.[11] The language has gone through several updates and extensions, but has remained static since 2008. For an excellent exposition of its features, see [481].

The complexity results in Section 2.5, and several other related results, are proved in [329]. The proofs are stated using classical representation, but can easily be translated to state-variable representation.

Ground representations. A classical representation is *ground* if it contains no unground atoms. With this restriction, the planning operators have no parameters; hence each planning operator represents just a single action. Ground classical representations usually are called *propositional* representations [194], because the ground atoms can be rewritten as propositional variables.

Every classical representation can be translated into an equivalent propositional representation by replacing each planning operator with all of its ground instances (all of the actions that it represents), but this incurs a combinatorial explosion in the size of the representation. If a planning operator has p parameters and each parameter has v possible values, then there are v^p ground instances. In a ground classical representation, each instance must be written explicitly, thus increasing the size of the representation by a multiplicative factor of v^p.

A *ground state-variable representation* is one in which all of the state variables are ground. Each ground state variable can be rewritten as a state variable that has no arguments (like an ordinary mathematical variable) [71, 490, 941]. Every state-variable representation can be translated into an equivalent ground state-variable representation, with a combinatorial explosion like the one in the classical-to-propositional conversion. If an action schema has p parameters and each parameter has v possible values, then the ground representation is larger by a factor of v^p.

[11] See https://www.icaps-conference.org/competitions/

The propositional and ground state-variable representation schemes are both PSPACE-equivalent [193, 70]. They can represent exactly the same set of planning problems as classical and state-variable representations; but as we just discussed, they may require exponentially more space to do so. This lowers the complexity class because computational complexity is expressed as a function of the size of the input.

In a previous work [410, Section 2.5.4], we claimed that propositional and ground state-variable representations could each be converted into the other with at most a linear increase in size, but that claim was only partially correct. Propositional actions can be converted to ground state-variable actions with at most a linear increase in size, using a procedure similar to the one we used to convert planning operators to action schemas. For converting in the reverse direction, the worst-case increase in size is polynomial but superlinear.[12]

The literature contains several examples of cases in which the problem representation and the computation of heuristic functions can be done more easily with ground state variables than with propositions [491, 941]. Helmert [490, Section 1.3] advances a number of arguments for considering ground state-variable representations superior to propositional representations.

2.7.2 Generalized Domain Models

If we ignore Remark 2.6, state-variable representation can be generalized to let states be arbitrary data structures, and an action schema's preconditions, effects, and cost be arbitrary computable functions operating on those data structures. Analogous generalizations can be made to the classical representation in Section 2.4 by allowing a predicate's arguments to be functional terms whose values are calculated procedurally rather than inferred logically [481]. Such generalizations make some kinds of planning algorithms and search heuristics inapplicable (see Section 3.6.7), but can make the domain models applicable to a much larger variety of application domains,

There are several other ways to generalize the action models in Section 2.3.2, such as explicit models of time requirements or multiple possible outcomes. Parts III, IV, V, and VI discuss several such generalizations.

2.7.3 Online Planning

The automated planning literature started very early to address the problems of integrating a planner in the acting loop of an agent. Concomitant to the seminal paper on STRIPS [359], Fikes [358] proposed a program called Planex for monitoring the execution of a plan and revising planning when needed, and our Reactive-Execution algorithm is inspired by the "triangle table" data structure used in that work. Numerous contributions followed (e.g., [36, 464, 1024, 1125, 909, 826, 177]). As we

[12]We believe it is a multiplicative factor in the interval $[\lg v, v]$, where v is the maximum size of any state variable's range. The lower bound, $\lg v$, follows from the observation that if there are n state variables, then representing the states may require $n \lg v$ propositions, with commensurate increases in the size of the planning operators. The upper bound, v, follows from the existence of a conversion procedure that replaces each action's effect $x(c_1, \ldots, c_n) \leftarrow d$ with the following set of literals:

$$\{p_x(c_1, \ldots, c_n, d)\} \cup \{\neg x(c_1, \ldots, c_n, d') \mid d' \in \mathit{Range}(x(c_1, \ldots, c_n)) \setminus \{d\}\}.$$

remarked at the beginning of Section 2.6.3, a limitation of all these works is that their soundness cannot be guaranteed if the environment changes too rapidly [62].

Problems involving integration of classical planning algorithms into the control architecture of specific systems, such as spacecraft, robots, or Web services, have been extensively studied. However, many of these contributions have assumed, as we did tacitly in Section 2.6, that the plans generated by the planning algorithms were directly executable, an assumption that is often unrealistic. Part V will discuss the integration of planning and acting with refinement of actions into commands, and ways to react to events.

The receding-horizon technique has been widely used in control theory, specifically in model-predictive control. The survey by Garcia et al. [380] traces its implementation back to the early sixties. The general idea is to use a predictive model to anticipate over a given horizon the response of a system to some control and to select the control such that the response has some desired characteristics. Optimal control seeks a response that optimizes a criterion. The use of these techniques together with task planning has been explored by Dean and Wellman [285].

Subgoaling has been used in the design of several problem-solving and search algorithms (e.g., [671, 641]). It is especially useful if the goals are *serialized*, that is, ordered in a sequence such that each one can be achieved without negating the ones that were previously achieved. A set of goals is *serializable* if they can be serialized [641],[13] and serializable goals can be further classified as *trivially* serializable (for example, goals that are fully independent) and *laboriously* serializable [86].

In practical applications, subgoaling often involves domain-specific techniques. For example, the video game *Killzone 2* [1137, 214] does subgoal planning with the planner running several times per second, concurrently with acting, for short-term objectives such as "get to shelter" for its computerized opponents.

Sampling techniques are widely used for handling stochastic models of uncertainty and nondeterminism (see Part III).

2.8 Exercises

2.1. Let $P_1 = (\Sigma, s_0, g_1)$ and $P_2 = (\Sigma, s_0, g_2)$ be two classical planning problems with the same planning domain and initial state. Let $\pi_1 = \langle a_1, \ldots, a_n \rangle$ and $\pi_2 = \langle b_1, \ldots, b_n \rangle$ be solutions for P_1 and P_2, respectively. Let $\pi = \langle a_1, b_1, \ldots, a_n, b_n \rangle$.

(a) If π is applicable in s_0, then is it a solution for P_1? For P_2? Why or why not?

(b) E_1 be the set of all state variables in $\text{eff}(a_1), \ldots, \text{eff}(a_n)$, and E_2 be the set of all state variables in $\text{eff}(b_1), \ldots, \text{eff}(b_n)$. If $E_1 \cap E_2 = \varnothing$, then is π applicable in s_0? Why or why not?

(c) Let P_1 be the set of all state variables that occur in $\text{pre}(a_1), \ldots, \text{pre}(a_n)$, and P_2 be the set of all state variables that occur in the preconditions of $\text{pre}(b_1), \ldots, \text{pre}(b_n)$. If $P_1 \cap P_2 = \varnothing$ and $E_1 \cap E_2 = \varnothing$, then is π applicable in s_0? Is it a solution for P_1? For P_2? Why or why not?

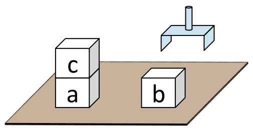
[13]For example, Figure 2.8 is a nonserializable planning problem called the Sussman anomaly [1145].

```
pickup(x)
    pre: loc(x) = table, top(x) = nil,
         holding = nil
    eff: loc(x) ← hand, holding ← x

putdown(x)
    pre: holding = x
    eff: loc(x) ← table, holding ← nil

unstack(x, y)
    pre: loc(x) = y, top(x) = nil,
         holding = nil
    eff: loc(x) ← hand, top(y) ← nil,
         holding ← x

stack(x, y)
    pre: holding = x, top(y) = nil
    eff: loc(x) ← y, top(y) ← x,
         holding ← nil
```

(a) action schemas, where $x, y \in$ *Blocks*

Objects = *Blocks* $\cup$ {hand, table, nil}

Blocks = {a, b, c}



s_0 = {top(a) = c, top(b) = nil,
top(c) = nil, holding = nil,
loc(a) = table, loc(b) = table,
loc(c) = a}

g = {loc(a) = b, loc(b) = c}

(b) objects, initial state, and goal

Figure 2.8. A blocks-world planning domain and a planning problem.

2.2. Give a classical planning problem P_1 and a solution π_1 for P_1 such that π_1 is minimal but not shortest. Give a classical planning problem P_2 and a solution π_2 for P_2 such that π_2 is acyclic but not minimal.

2.3. Let $\Sigma = (S, A, \gamma)$ be the state-transition system represented by $\mathcal{H}$, R X, and $\mathcal{A}$ in Examples 2.1 and 2.8.

(a) How many states are in S? How many actions are in A? Briefly describe them.

(b) Let S' be the set of all states reachable from s_0, that is,

$$S' = \{\gamma(s_0, \pi) \mid \pi \text{ is a plan that is applicable in } s_0\}.$$

How many states are in S'? Give an example of a state in S that is not in S'.

(c) Do the states in S all have sensible meanings? Do the states in S'?

(d) Let $P = (\Sigma, s_0, g)$, where s_0 is as in Example 2.1 and g = {pos(c1) = d2}. Give a shortest solution for P. Give a solution that is minimal but not shortest. How many minimal solutions are there?

2.4. The *blocks world* is a well-known classical planning domain[14] in which a set of cubical blocks, *Blocks* = {a, b, c, ...}, are arranged in stacks of varying size on an infinitely large table, table. To move the blocks, there is a robot hand, hand, that can hold at most one block at a time.

Figure 2.8(a) gives the action schemas. For each block x, loc(x) is x's location, which may be table, hand, or another block; and top(x) is the block (if any) that is on x, with top(x) = nil if nothing is on x. Finally, holding tells what block the robot hand is holding, with holding = nil if the hand is empty.

[14]More accurately, because the number of blocks may vary, it is a set of planning domains.

(a) Why are there four action schemas rather than just two?
(b) Is the state variable holding really needed? Why or why not?
(c) In the planning problem in Figure 2.8(b), how many states satisfy g?
(d) Give necessary and sufficient conditions for a set of blocks-world atoms to be a state.
(e) Is every blocks world planning problem solvable? Why or why not?

2.5. This exercise involves the **for** loop at Line 2 of Reactive-Execution.

(a) Give the loop's big-O time complexity as a function of n and k, where n is the number of actions in π, and k is the maximum number of preconditions and effects of each action in π.
(b) The **for** loop can be made much faster by using a table that relates each action's preconditions to effects of previous actions and the initial state, and relates each action's effects to the preconditions of subsequent actions and the goal. Write such a data structure, rewrite the **for** loop to use it, and analyze the resulting time complexity.

2.6. Suppose an actor starts in state s_0 of the planning problem shown in Figure 2.5, using Run-Lazy-Lookahead with a *Lookahead* algorithm that always returns the shortest possible solution plan. The first call to *Lookahead* returns

$$\pi = \{\text{take(r1,c1,loc1), move(r1,loc1,loc2), put(r1,c1,loc2)}\}.$$

(a) Suppose that after the actor has performed take(r1,c1,loc1) and move(r1,loc1,loc2), monitoring reveals that c1 fell off of the robot and is still back at loc1. Tell what will happen, step by step. Assume that *Lookahead*(P) will always return the best solution for P.
(b) Repeat part (a) assuming that c1 will fall off of the robot every time it performs move(r1,loc1,loc2).
(c) Repeat part (a) using Run-Lookahead.
(d) Suppose that after the actor has performed take(r1,c1,loc1), monitoring reveals that r1's wheels have stopped working, hence r1 cannot move from loc1. What should the actor do to recover? How would you modify Run-Lazy-Lookahead and Run-Lookahead to accomplish this?

2.7. Consider the planning domain in Examples 2.1 and 2.8.

(a) Rewrite the planning domain using classical representation.
(b) Rewrite it in PDDL.

3 Planning with Deterministic Models

Section 1.1.2 introduced the idea of domain-independent planning algorithms. Domain-independent classical-planning algorithms, which are the subject of this chapter, were until recently the most widely studied class of AI planning algorithms.

This chapter is organized as follows. Section 3.1 classifies and describes a variety of forward-search planning algorithms, and Section 3.2 provides some heuristics to guide such algorithms. Sections 3.3 and 3.4 describe backward search and plan-space planning algorithms. Section 3.6 provides discussion and bibliographic notes, and Section 3.7 contains exercises.

3.1 Forward State-Space Search

```
Forward-Search($\Sigma, s_0, g$)
    $s \leftarrow s_0$;  $\pi \leftarrow \langle\rangle$
    while $s \not\models g$ do
        if $Applicable(s) = \varnothing$ then return failure
1       nondeterministically choose $a \in Applicable(s)$
        $s \leftarrow \gamma(s, a)$;  $\pi \leftarrow \pi \cdot a$
    return $\pi$
```

Algorithm 3.1. Forward-Search, a schema for forward state-space search.

Given a planning problem $P = (\Sigma, s_0, g)$, many classical planning algorithms search forward from the initial state s_0 to try to construct a sequence of actions that reaches a state in S_g. Algorithm 3.1, Forward-Search, is a procedural schema for a wide variety of such algorithms. In Line 1, the idea is to try various actions $a \in Applicable(s)$ until we find one that we like, and the "nondeterministically choose" command is an abstraction that allows us to ignore the precise order in which to try them. This lets us describe properties of all algorithms that search the same search space, irrespective of the order in which they visit the nodes. For more details of such *nondeterministic algorithms*, see Appendix A.

The *search space* for a planning problem $P = (\Sigma, s_0, g)$ is a graph containing every path that Forward-Search(Σ, s_0, g) can generate, that is, all paths that start at s_0 and do not continue beyond goal states. We will use the following terminology and notation:

- To keep the presentation simple, we will write each node as a pair $\nu = (\pi, s)$, where π is a plan and $s = \gamma(s_0, \pi)$. In practical implementations, however, ν will usually include other information such as its depth, cost, and pointers to

Free pre-publication, for personal use only. To be published by Cambridge University Press.

parent and child nodes. Most implementations will not store π explicitly in v, but instead will calculate it by tracing the "parent" pointers from v back to the initial node.

- The *initial* or *starting* node is the pair $(\langle\rangle, s_0)$, where $\langle\rangle$ is the empty plan and s_0 is the initial state.
- If $v = (\pi, s)$ is a node and $a \in A$ is applicable in s, then the node $(\pi \cdot a, \gamma(s, a))$ is a *child* of v. To *expand* a node v means to generate all of its children.
- A *successor* or *descendant* of a node v is any child of v or, recursively, a successor of any child of v. An *ancestor* of v is any node v' such that v is a successor of v'.
- The *depth* of a node $v = (\pi, s)$ is the length of the path $\widehat{\gamma}(s_0, \pi)$, or equivalently, the length of π. The search space's *height* is the length of the longest acyclic path that starts at the initial node. Its *maximum branching factor* is the maximum number of children of any node.
- The *cost* of a node $v = (\pi, s)$ is $\text{cost}(v) = \text{cost}(\pi)$.

```
Forward-Search-Det($\Sigma, s_0, g$)
    $Frontier \leftarrow \{(\langle\rangle, s_0)\}$          // $(\langle\rangle, s_0)$ is the initial node
    $Expanded \leftarrow \varnothing$
    while $Frontier \neq \varnothing$ do
1       select a node $(\pi, s) \in Frontier$
        remove $(\pi, s)$ from $Frontier$ and add it to $Expanded$
2       if $s$ satisfies $g$ then return $\pi$
        $Children \leftarrow \{(\pi \cdot a, \gamma(s, a)) \mid a \in A$ is applicable in $s\}$
3       prune (i.e., select and remove) 0 or more nodes from $Children$, $Frontier$,
          and $Expanded$
4       $Frontier \leftarrow Frontier \cup Children$
    return failure
```

Algorithm 3.2. Forward-Search-Det, a deterministic version of Forward-Search.

Algorithm 3.2, Forward-Search-Det, is a deterministic version of Forward-Search in which *Frontier* is a set of nodes that are candidates to be visited, and *Expanded* is a set of nodes that have already been visited. During each loop iteration, the algorithm selects a node, generates its children, prunes some unpromising nodes, and updates *Frontier* to include the remaining children. Many forward-search planning algorithms can be described as versions of Forward-Search-Det by specifying how they select nodes in Line 1 and prune nodes in Line 3.

In many forward-search algorithms, the pruning step (Line 3 of Forward-Search-Det) often includes a *cycle-checking* step:

> Remove from *Children* every node $v = (\pi, s)$ for which an ancestor of v has the same state s.

In classical planning problems (and any other planning problems in which the state space is finite), cycle checking guarantees that the search will always terminate.

3.1.1 Breadth-First and Depth-First Search

Breadth-first search can be written as a version of Forward-Search-Det with selection and pruning as follows:

- *Node selection.* Select a node $v = (\pi, s) \in Children$ that minimizes the length of π. If there are several such nodes, some possible tie-breaking rules are to choose the leftmost node or to choose a node that minimizes $\text{cost}(\pi)$, $h(s)$, or $f(v)$.
- *Pruning.* Remove from *Children* and *Frontier* every node (π, s) such that *Expanded* contains a node (π', s') such that $s' = s$. This keeps the algorithm from expanding s more than once.

In classical planning problems, breadth-first search will always terminate and will return a solution if one exists. The solution will be shortest but not necessarily cost-optimal.

Because breadth-first search keeps only one path to each node, its worst-case memory requirement is $O(|S|)$, where $|S|$ is the number of nodes in the search space. Its worst-case running time is $O(b|S|)$, where b is the maximum branching factor.

Depth-first search is usually written as a recursive algorithm, but it can be rewritten to run iteratively as a version of Forward-Search-Det In classical planning problems, it will always terminate and will return a solution if one exists, but the solution will not necessarily be shortest or cost-optimal.

Depth-first search only needs to remember the nodes along the current path and the children of those nodes, so the worst-case memory requirement is $O(bl)$, where b is the maximum branching factor and l is the height of the search space. However, the worst-case running time is $O(b^l)$, which can be much worse than $O(|S|)$ if there are many paths from s_0 to each state.

3.1.2 Greedy Search

Greedy search is a depth-first search with no backtracking:

- *Node selection.* Select a node $(\pi, s) \in Children$ that minimizes $h(s)$.
- *Pruning.* First, do cycle checking. Then assign $Frontier \leftarrow \varnothing$, so that Line 4 of Forward-Search-Det will be the same as assigning $Frontier \leftarrow Children$.

The search follows a single path, and prunes all nodes not on that path. It is guaranteed to terminate on classical planning problems, but it is not guaranteed to return an optimal solution or even a solution at all. Its worst-case running time is $O(bl)$ and its the worst-case memory requirement is $O(l)$, where l is the height of the search space and b is the maximum branching factor.

3.1.3 Uniform-Cost Search

Uniform-cost (or *least-cost first*) search is somewhat like breadth-first search, but it does node selection and pruning using the accumulated cost of each node:

- *Node selection.* Select a node $(\pi, s) \in Children$ that minimizes $\text{cost}(\pi)$.

- *Pruning*. Remove from *Children* and *Frontier* every node (π, s) such that *Expanded* contains a node (π', s). In classical planning problems (and any other problems in which all costs are nonnegative), it can be proved that $\text{cost}(\pi') \leq \text{cost}(\pi)$, so this step ensures that the algorithm only keeps the least costly path to each node.

In classical planning problems, the search is guaranteed to terminate and to return an optimal solution. Like breadth-first search, its worst-case running time and memory requirement are $O(b|S|)$ and $O(|S|)$, respectively.

3.1.4 Using a Heuristic Function

Most forward-search planning algorithms attempt to find a solution without exploring the entire search space, which in the worst case can be exponentially large.[1] To make informed guesses about which parts of the search space are more likely to lead to solutions, node selection (Line 1 of Forward-Search-Det) often involves a *heuristic function* $h : S \rightarrow \mathbb{R}$ that returns an estimate of $h^*(s)$, which is the minimum cost of getting from s to a goal state:

$$h(s) \approx h^*(s) = \min\{\text{cost}(\pi) \mid \gamma(s, \pi) \text{ satisfies } g\}. \tag{3.1}$$

As a special case, we require that $h(s) = 0$ whenever s satisfies g. In Section 3.2 we will discuss some ways to compute heuristic functions.

Given a node $v = (\pi, s)$, some forward-search algorithms will use h to compute an estimate $f(v)$ of the minimum cost of any solution plan that begins with π:[2]

$$f(v) = \text{cost}(\pi) + h(s) \approx f^*(v), \tag{3.2}$$

where

$$f^*(v) = \min\{\text{cost}(\pi \cdot \pi') \mid \gamma(s_0, \pi \cdot \pi') \text{ satisfies } g\}. \tag{3.3}$$

If $0 \leq h(s) \leq h^*(s)$ for every $s \in S$, then h is *admissible*, from which the following properties follow immediately:

- $f(v) \leq f^*(v)$, i.e., $f(v)$ is a lower bound on the cost of every solution that begins with π;
- If s is a goal state, then $h(s) = 0$ and $f(v) = \text{cost}(\pi)$.

3.1.5 A*

A*, Algorithm 3.3, is similar to uniform-cost search but it uses a heuristic function for node selection. This makes A*'s pruning more complicated. If a node in *Children* has the same state as one in *Expanded* or *Frontier*, it compares their costs and keeps only the least costly one.

Here are some of A*'s properties:

[1]The worst-case computational complexity is EXPSPACE-equivalent (see Section 3.6), although the complexity of a specific planning domain usually is much less.

[2]In many presentations of heuristic search, $f(v)$ is written as $f(s)$, but that causes ambiguity if there is more than one plan π such that $\gamma(s_0, \pi) = s$. Making f a function of v avoids that difficulty.

```
A*(Σ, $s_0$, $g$)
    Use Forward-Search-Det with node selection and pruning as follows:
    • Node selection. In Line 1, select a node $v \in Children$ that minimizes
      $f(v)$ (defined in Equation 3.2).
    • Pruning. In Line 3, for each node $(\pi, s) \in Children$, if there is a node
      $(\pi', s') \in Frontier \cup Expanded$ such that $s' = s$, then keep whichever
      of $(\pi, s)$ and $(\pi', s')$ has lower cost,[a] and prune the other one and its
      descendants.
```

[a]If both nodes have the same cost, a typical tie-breaking rule is to keep the oldest one.

Algorithm 3.3. The A* algorithm.

- *Termination, completeness, and optimality.* On any classical planning problem, A* will terminate and return a solution if one exists. If h is admissible, then this solution will be optimal.
- *Epsilon-optimality.* If h is *ϵ-admissible* (i.e., if there is an $\epsilon > 0$ such that $0 \le h(s) \le h^*(s) + \epsilon$ for every $s \in S$), then the solution returned by A* will be within ϵ of optimal.
- *Monotonicity.* If $h(s) \le \text{cost}(\gamma(s, a)) + h(\gamma(s, a))$ for every state s and applicable action a, then h is said to be *monotone* or *consistent*. In this case, $f(v) \le f(v')$ for every child v' of a node v, from which it can be shown that A* will never prune any nodes from *Expanded*, and will expand no state more than once.
- *Informedness.* Let h_1 and h_2 be admissible heuristic functions such that h_2 *dominates*[3] h_1, i.e., $0 \le h_1(s) \le h_2(s) \le h^*(s)$ for every $s \in S$. Then A* will never expand more nodes[4] with h_2 than with h_1, and in most cases, it will expand fewer nodes with h_2 than with h_1.

A*'s primary drawback is its space requirement: it needs to store every state that it visits. Like uniform-cost search, A*'s worst-case running time and memory requirement are $O(b|S|)$ and $O(|S|)$. However, with a good heuristic function, A*'s running time and memory requirement are usually much smaller.

3.1.6 Greedy Best-First Search

For classical planning problems where nonoptimal solutions are acceptable, the most frequently used search algorithm is *Greedy Best-First Search* (GBFS), which works as follows:

Like hill climbing, GBFS continues to expand nodes along its current path as long as that path looks promising. But like A*, GBFS stores every state that it visits. Hence

[3]Dominance has often been described by saying that "h_2 is *more informed* than h_1," but that phrase is awkward because h_2 always dominates itself.

[4]This assumes that when A* chooses among nodes that have the same f-value, it always uses the same tie-breaking rule.

GBFS(Σ, s_0, g)

Use Forward-Search-Det with node selection and pruning as follows:

- *Node selection.* In Line 1, select a node $(\pi, s) \in$ *Frontier* that minimizes $h(s)$.
- *Pruning.* In Line 3, pruning should at least include cycle checking. In other cases where a node in $(\pi, s) \in$ *Children* has the same state s as some other node, one could prune one of the nodes arbitrarily, prune the higher-cost node, or do no pruning.[a]

[a]The rationale for no pruning is that with a good heuristic function, GBFS is unlikely to select both nodes for expansion.

Algorithm 3.4. GBFS, greedy best-first search.

it can easily switch to a different path if the current path dead-ends or ceases to look promising.

Like A*, GBFS's worst-case running time and memory requirement are $O(b|S|)$ and $O(|S|)$. Unlike A*, GBFS is not guaranteed to return optimal solutions; but in most cases, it will explore far fewer paths than A* and find solutions much more quickly.

```
DFBB($\Sigma, s_0, g$)
    return (DFBB1($\Sigma, (s_0, \langle\rangle), g$, failure, $\infty$))

DFBB1($\Sigma, \nu, g, \pi^*, c^*$)
    $(\pi, s) \leftarrow \nu$
    if $s \models g$ and cost$(\pi) < c^*$ then
1       $c^* \leftarrow$ cost$(\pi)$;  $\pi^* \leftarrow \pi$
2   else if $f(\nu) < c^*$ then
        Children $\leftarrow \{(\pi \cdot a, \gamma(s, a)) \mid a \in A$ is applicable in $s\}$
        foreach $\nu \in$ Children do
            $(c^*, \pi^*) \leftarrow$ DFBB1($\Sigma, \nu, g, \pi^*, c^*$)
    return $(c^*, \pi^*)$
```

Algorithm 3.5. DFBB, depth-first branch and bound.

3.1.7 Depth-First Branch and Bound

Depth-first branch and bound, DFBB, is a modified version of depth-first search. If the heuristic function f in Line 2 is admissible, it will return a least-cost solution if a solution exists. In the initial recursive call, $\nu = (s_0, \langle\rangle)$ is the root node. The variables π^* and c^* are the least costly solution seen so far and its cost, which are updated in Line 1 each time a solution is found. Line 2 expands ν only if $f(\nu) < c^*$, which can prune large parts of the search space if c^* is small. When the recursive calls are finished, DFBB returns the pair (π^*, c^*).

DFBB has the same termination, completeness, and optimality properties as A*. The only nodes in its recursion stack are the nodes in the current path and their sibling nodes, so its memory requirement is usually much lower than A*'s. However, like depth-first search, if there are many paths to a state it may revisit the state many times, which can make its running time much worse than A*'s. In the worst case, its running time and memory requirement are $O(b^l)$ and $O(bl)$, the same as for depth-first search.

```
IDS($\Sigma, s_0, g$)
    **for** $k \leftarrow 1$ **to** $\infty$ **do**
        do a depth-first search of $(\Sigma, s_0, g)$, backtracking at all nodes of depth $k$
        **if** the search found a solution **then return** it
        **if** the search generated no nodes of depth $k$ **then return** failure
```

Algorithm 3.6. IDS, iterative-deepening search.

3.1.8 Iterative Deepening

Several forward-search algorithms wrap a depth-first search inside an iterative loop. One of the best known is *iterative-deepening search* (IDS), Algorithm 3.6. On classical planning problems, its termination, completeness, and optimality properties are the same as for breadth-first search. Its primary advantage over breadth-first search is that its worst-case memory requirement is only $O(bd)$, where d is the depth of the solution returned if there is one, or the height of the search space otherwise. If the number of nodes at each depth k grows exponentially with k, then IDS's worst-case running time is $O(b^d)$, which can be substantially worse than breadth-first search if there are many paths to each state.

```
IDA*($\Sigma, s_0, g$)
    $c \leftarrow 0$
    **while** True **do**
        do a depth-first search of $(\Sigma, s_0, g)$, backtracking whenever $f(v) > c$
        **if** the search found a solution **then return** it
        **if** the search did not generate an $f(v) > c$ **then return** failure
        $c \leftarrow$ the smallest $f(v) > c$ where backtracking occurred
```

Algorithm 3.7. IDA*, iterative-deepening A*.

A closely related algorithm, *iterative-deeping A** (Algorithm 3.7), uses a cost bound rather than a depth bound. On classical planning problems, IDA*'s termination, completeness, and optimality properties are the same as those of A*, and its worst-case memory requirement is $O(bl)$, where l is the height of the search space. If the number of nodes grows exponentially with c (which usually is true in classical planning problems but less likely to be true in nonclassical ones), then IDA*'s worst-case running time is $O(b^d)$, where d is either the depth of the solution found by IDA*, or the height of the search space if there is no solution. This can be substantially worse than A*'s running time if there are many paths to each state.

3.1.9 Choosing a Forward-Search Algorithm

It is difficult to give any hard-and-fast rules for choosing among the forward-search algorithms presented here, but here are some rough guidelines.

If the solution must be optimal (or within ϵ of optimal) and one has a good heuristic function that is admissible (or ϵ-admissible, respectively), then an A*-like algorithm is a good choice if the state space is small enough to store every node in main memory. If the state space is too large to hold in main memory, then an algorithm such as DFBB or IDA* may be worth trying, but there may be problems with excessive running time if the state space has many paths to each state.

If a nonoptimal solution is acceptable and a good heuristic function is available, often the best choice is to develop a planning algorithm that uses either GBFS or one that weights h more heavily than g. There are no guarantees as to GBFS's performance, but with a good heuristic function it usually works quite well.

For integration of planning into acting, an important question is how to turn any of these algorithms into online algorithms. This is discussed further in Chapter 3.

3.2 Heuristic Functions

Recall from Section 3.1.4 that a heuristic function h computes an estimate of $h^*(s)$, and h is admissible if $0 \le h(s) \le h^*(s)$ for every state s.

The simplest possible heuristic function is $h(s) = 0$ for every s. This is admissible and is trivial to compute, but provides no useful information. Usually we will want a better estimate of h^*. If it can be computed in a polynomial amount of time and can provide an exponential reduction in the number of nodes examined by the planning algorithm, this makes the computational effort worthwhile.

The best-known way to produce heuristic functions is *relaxation*. To relax a planning domain $\Sigma = (S, A, \gamma, \text{cost})$ and planning problem $P = (\Sigma, s_0, g)$ is to change them by making actions more widely applicable and introducing additional states, actions, plans, and goals. This produces a *relaxed* planning domain $\Sigma' = (S', A', \gamma', \text{cost})$ and planning problem $P' = (\Sigma', s'_0, g')$ having the following property: if π is any solution for P, then P' has a solution π' such that $\text{cost}(\pi') \le \text{cost}(\pi)$.

Given an algorithm for solving planning problems in Σ', we can use it to create a heuristic function for P that works as follows: given a state $s \in S$, use the algorithm to solve (Σ', s, g'), and return the cost of the solution. If the algorithm always finds optimal solutions, then the heuristic function will be admissible.

Example 3.1. Figure 3.1 depicts a road network that connects a set of locations, each represented by a pair of coordinates. Let us say that two locations are *adjacent* if there is a road between them. Suppose a robot can move from a location (x, y) to an

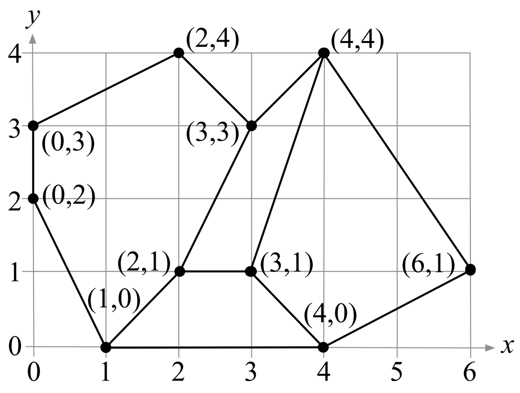


Figure 3.1. A network of locations connected by roads.

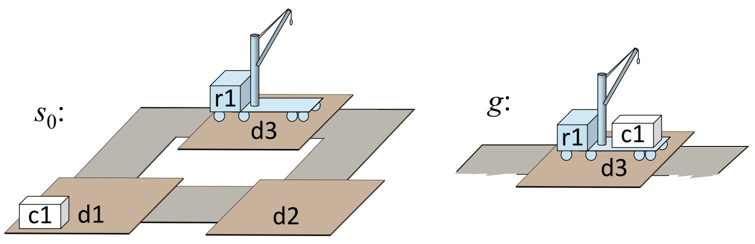


Figure 3.2. Initial state and goal for Example 3.2.

adjacent location (x', y') at a cost equal to the distance from (x, y) to (x', y'). The road network and the movement actions can be represented as a planning domain in which each state $s_{x,y}$ is represented by the (x, y) coordinates of the robot's current location.

Suppose we want to plan a sequence of move actions to move the robot from (x, y) to location $(6, 1)$ at the minimum possible cost. One possible heuristic function is the Euclidean distance,

$$h(s_{x,y}) = \sqrt{(x-6)^2 + (y-1)^2}, \tag{3.4}$$

which is the length of an optimal solution for a relaxed problem in which the actor is not constrained to follow roads. This is a lower bound on the cost of every route to location $(6, 1)$, so h is admissible. □

The preceding heuristic function is domain-specific, but there are many *domain-independent* heuristic functions that can be used in any classical planning domain. The following subsections describe a few of them.

Although the planning algorithms earlier in this chapter did not require any particular domain representation, the heuristic functions do. We will use the state-variable representation described in Section 2.3, and we will use the following planning problem as a running example.

Example 3.2. Figure 3.2 shows a planning problem $P = (\Sigma, s_0, g)$, in a simplified version of the planning domain in Example 2.1. The objects include one robot r1, one container c1, three docks d1, d2, d3, no piles, and the constant nil. There are no rigid relations. The state variables are cargo(r1), loc(c1), and loc(r1), with

$$\begin{aligned} \mathit{Range}(\mathsf{cargo}(\mathsf{r1})) &= \{\mathsf{c1}, \mathsf{nil}\}; \\ \mathit{Range}(\mathsf{loc}(\mathsf{c1})) &= \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}, \mathsf{r1}\}; \\ \mathit{Range}(\mathsf{loc}(\mathsf{c1})) &= \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}\}. \end{aligned}$$

Here are the action schemas and their parameter ranges:

take(r, c, l)
  pre: cargo(r) = nil, loc$(c) = l$, loc$(r) = l$
  eff: cargo$(r) \leftarrow c$, loc$(c) \leftarrow r$
  cost: 1

move(r, d, e)
  pre: loc$(r) = d$
  eff: loc$(r) \leftarrow e$
  cost: 1

put(r, c, l)
  pre: cargo$(r) = c$, loc$(r) = l$
  eff: cargo$(r) \leftarrow$ nil, loc$(c) \leftarrow l$
  cost: 1

$\mathit{Range}(c) = \{\mathsf{c1}\}$;
$\mathit{Range}(d) = \mathit{Range}(e) = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}\}$;
$\mathit{Range}(l) = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}, \mathsf{r1}\}$;
$\mathit{Range}(r) = \{\mathsf{r1}\}$.

P's initial state and goal are

$$\begin{aligned} s_0 &= \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d3}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{nil}, \mathsf{loc}(\mathsf{c1}) = \mathsf{d1}\}; \\ g &= \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d3}, \mathsf{loc}(\mathsf{c1}) = \mathsf{r1}\}. \end{aligned}$$

In s_0, the applicable actions are $a_1 = \mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d1})$ and $a_2 = \mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d2})$. Let

$$\begin{aligned} s_1 &= \gamma(s_0, a_1) = \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{nil}, \mathsf{loc}(\mathsf{c1}) = \mathsf{d1}\}, \\ s_2 &= \gamma(s_0, a_2) = \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d2}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{nil}, \mathsf{loc}(\mathsf{c1}) = \mathsf{d1}\}. \end{aligned}$$

If we run GBFS on P, then in Line 1 of Forward-Search-Det, GBFS will choose between a_1 and a_2 by evaluating $h(s_1)$ and $h(s_2)$. The following subsections describe several possibilities for what h might be. □

3.2.1 Delete-Relaxation Heuristics

Several heuristic functions are based on the notion of *delete-relaxation*, in which applying an action never removes old atoms from a state, but simply adds new ones.

If a state s includes an atom $x = v$ and an applicable action a has an effect $x \leftarrow w$, then the delete-relaxed result of applying a will be a "state" $\gamma^+(s, a)$ that includes both $x = v$ and $x = w$. We will make the following definitions:

- A *relaxed state* (or *r-state*, for short) is any set $\hat{s}$ of ground atoms that contains at least one occurrence of every state variable $x \in X$. It follows that every state is also an r-state.
- A relaxed state $\hat{s}$ *r-satisfies* a set of literals g if S contains a subset $s \subseteq \hat{s}$ that satisfies g.
- An action a is *r-applicable* in an r-state $\hat{s}$ if $\hat{s}$ r-satisfies pre(a). In this case, the resulting r-state is

$$\gamma^+(\hat{s}, a) = \hat{s} \cup \gamma(s, a). \tag{3.5}$$

- By extension, a plan $\pi = \langle a_1, \ldots, a_n \rangle$ is r-applicable in an r-state $\hat{s}_0$ if there are r-states $\hat{s}_1, \ldots, \hat{s}_n$ such that

$$\hat{s}_1 = \gamma^+(\hat{s}_0, a_1),\ \hat{s}_2 = \gamma^+(\hat{s}_1, a_2),\ \ldots,\ \hat{s}_n = \gamma^+(\hat{s}_{n-1}, a_n).$$

  In this case, $\gamma^+(\hat{s}_0, \pi) = \hat{s}_n$.
- A plan π is a *relaxed solution* for a planning problem $P = (\Sigma, s_0, g)$ if $\gamma^+(s_0, \pi)$ r-satisfies g. Thus the cost of the optimal relaxed solution is

$$\Delta^+(s, g) = \min\{\mathrm{cost}(\pi) \mid \gamma^+(s, \pi) \text{ r-satisfies } g\}.$$

For a planning problem $P = (\Sigma, s_0, g)$, the *optimal relaxed solution* heuristic is

$$h^+(s) = \Delta^+(s, g).$$

Example 3.3. Let P be the planning problem in Example 3.2. Let $\hat{s}_1 = \gamma^+(s_0, \mathsf{move(r1, d3, d1)})$ and $\hat{s}_2 = \gamma^+(\hat{s}_1, \mathsf{take(r1, c1, d1)})$. Then

$$\begin{aligned}\hat{s}_1 &= \{\mathsf{loc(r1) = d1, loc(r1) = d3, cargo(r1) = nil, loc(c1) = d1}\};\\ \hat{s}_2 &= \{\mathsf{loc(r1) = d1, loc(r1) = d3, cargo(r1) = nil, cargo(r1) = c1,}\\ &\quad \mathsf{loc(c1) = d1, loc(c1) = r1}\}.\end{aligned}$$

Because $\hat{s}_2$ r-satisfies g, the plan $\pi = \langle \mathsf{move(r1, d3, d1), take(r1, c1, d1)} \rangle$ is a relaxed solution for P. There are no shorter relaxed solutions, so $h^+(s) = \Delta^+(s_0, g)$. □

Every ordinary solution for P is also a relaxed solution for P, so $h^+(s) \leq h^*(s)$ for every s. Thus h^+ is admissible, so it can be used with A* to find an optimal solution for P. However, h^+ is expensive to compute: the problem of finding an optimal relaxed solution for a planning problem P is NP-hard.[5]

3.2.2 Relaxed Planning Graph Heuristics

We now describe an approximation to h^+ that is easier to compute. It is based on the fact that if A is a set of actions that are all r-applicable in a relaxed state $\hat{s}$, then they will produce the following r-state regardless of the order in which they are applied:

$$\gamma^+(\hat{s}, A) = \hat{s} \cup \bigcup_{a \in A} \mathrm{eff}(a). \tag{3.6}$$

HFF, Algorithm 3.8, starts at an initial r-state $\hat{s}_0 = s$, and uses Equation 3.6 to generate a sequence of successively larger r-states and sets of r-applicable actions,

$$\hat{s}_0, A_1, \hat{s}_1, A_2, \hat{s}_2 \ldots,$$

until it generates an r-state that r-satisfies g. From this sequence, HFF extracts a relaxed solution and returns its cost. In Line 2, if the sequence has converged to an r-state that does not r-satisfy g, then the planning problem has no solution.

[5]The problem is NP-complete when P is ground [133], hence is at least NP-hard when P is lifted.

```
HFF($\Sigma, s, g$)
    $\hat{s}_0 = s$;  $A_0 = \varnothing$
1   for $k \leftarrow 1$ by 1 until a subset of $\hat{s}_k$ r-satisfies $g$ do
        $A_k \leftarrow$ {all actions that are r-applicable in $\hat{s}_{k-1}$}
        $\hat{s}_k \leftarrow \gamma^+(\hat{s}_{k-1}, A_k)$
2       if $\hat{s}_k = \hat{s}_{k-1}$ then return $\infty$          // $(\Sigma, s, g)$ has no solution
    $\hat{g}_k \leftarrow g$
3   for $i \leftarrow k$ down to 1 do
        arbitrarily choose a minimal set of actions $\hat{a}_i \subseteq A_i$ such that $\gamma^+(\hat{s}_i, \hat{a}_i)$
          satisfies $\hat{g}_i$
        $\hat{g}_{i-1} \leftarrow (\hat{g}_i - \mathrm{eff}(\hat{a}_i)) \cup \mathrm{pre}(\hat{a}_i)$
4   $\hat{\pi} \leftarrow \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_k \rangle$
    return $\sum\{\mathrm{cost}(a) \mid a$ is an action in $\hat{\pi}\}$
```

Algorithm 3.8. HFF, which computes the Fast-Forward heuristic.

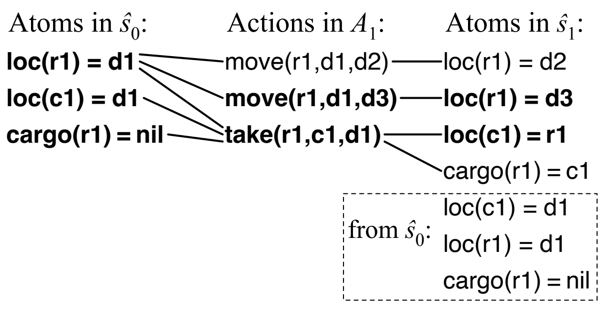


Figure 3.3. Computation of HFF$(\Sigma, s_1, g) = 2$. The solid lines indicate the actions' preconditions and effects. The elements of $\hat{g}_0$, $\hat{a}_1$, and $\hat{g}_1$ are shown in boldface.

The *Fast-Forward* heuristic[6] is

$$h^{\mathrm{FF}}(s) = \text{ the value returned by HFF.} \tag{3.7}$$

The definition of h^{FF} is ambiguous, because the returned value may vary depending on HFF's choices of $\hat{a}_k, \hat{a}_{k-1}, \ldots, \hat{a}_1$ in the loop at Line 3. Furthermore, because there is no guarantee that these choices are the optimal ones, h^{FF} is not admissible.

The running time for HFF is nontrivial, but it is polynomial in the total number of actions and ground atoms in the planning domain.

Example 3.4. In Example 3.2, suppose GBFS's heuristic function is h^{FF}. To compute $h^{\mathrm{FF}}(s_1)$, HFF begins with $\hat{s}_0 = s_1$, and computes A_1 and $\hat{s}_1$ in the loop at Line 1. Figure 3.3 illustrates the computation: the lines to the left of each action show which atoms in $\hat{s}_0$ satisfy its preconditions, and the lines to the right of each action show which atoms in $\hat{s}_1$ are its effects. For the loop at Line 3, HFF begins with $\hat{g}_1 = g$ and

[6]The name comes from the FF planner in which this heuristic was introduced; see Section 3.6.5.

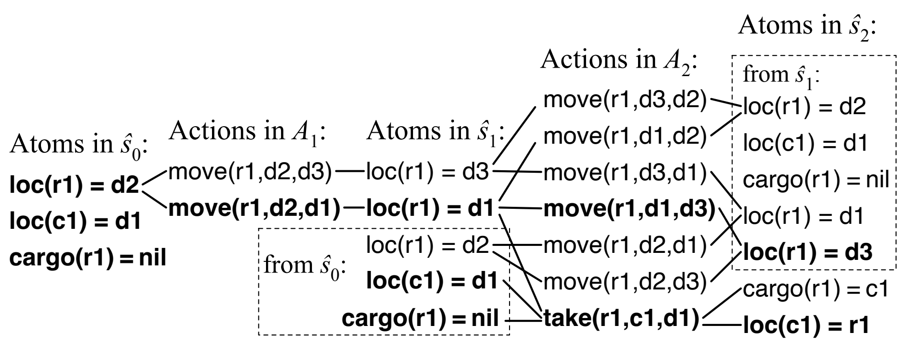


Figure 3.4. Computation of HFF$(\Sigma, s_2, g) = 3$. The atoms and actions in each $\hat{g}_i$ and $\hat{a}_i$ are shown in boldface, and the solid lines indicate their preconditions and effects.

computes $\hat{a}_1$ and $\hat{g}_0$; these sets are shown in boldface in Figure 3.3. In Line 4, the relaxed solution is

$$\hat{\pi} = \langle \hat{a}_1 \rangle = \langle \{\text{move}(\text{r1}, \text{d1}, \text{d3}), \text{take}(\text{r1}, \text{c1}, \text{d1})\} \rangle.$$

Thus HFF returns $h^{\text{FF}}(s_1) = \text{cost}(\hat{\pi}) = 2$.

Figure 3.4 is a similar illustration of HFF's computation of $h^{\text{FF}}(s_2)$. For the loop at Line 1, HFF begins with $\hat{s}_0 = s_2$ and computes the sets A_1, $\hat{s}_1$, A_2, and $\hat{s}_2$. For the loop at Line 3, HFF begins with $\hat{g}_2 = g$ and computes $\hat{a}_2$, $\hat{g}_1$, $\hat{a}_1$, and $\hat{g}_0$, which are shown in boldface in Figure 3.4. In Line 4, the relaxed solution is

$$\hat{\pi} = \langle \hat{a}_1, \hat{a}_2 \rangle = \langle \{\text{move}(\text{r1}, \text{d2}, \text{d1})\}, \{\text{move}(\text{r1}, \text{d1}, \text{d3}), \text{take}(\text{r1}, \text{c1}, \text{d1})\} \rangle,$$

so HFF returns $h^{\text{FF}}(s_2) = \text{cost}(\hat{\pi}) = 3$.

Thus $h^{\text{FF}}(s_1) < h^{\text{FF}}(s_2)$, so GBFS will choose to expand s_1 next. □

The graph structures in Figures 3.3 and 3.4 are called *relaxed planning graphs*.

Improving HFF. If s is the current state and A is the set of applicable actions, then it is not very efficient to call HFF$(\gamma(s, a))$ once for each $a \in A$. The calls to HFF are likely to produce planning graphs that have a large amount of overlap, thus incurring a lot of repeated effort. Second, there may be many actions that will not appear in any non-redundant solution plan, and calling HFF on them is wasted effort. The same result can be produced more efficiently as follows. First, call HFF(s) to produce a single planning graph G. Second, prune from A all *non-helpful* actions, that is, actions that do not appear in any relaxed solution. Third, for each non-pruned action $a \in A$, extract from G a relaxed solution and h^{FF} value for $\gamma(s, a)$. Similar approaches can be devised for the heuristic functions in the following sections.

3.2.3 Landmark Heuristics

Let $P = (\Sigma, s, g)$ be a planning problem, and $\phi = p_1 \vee \ldots \vee p_m$ be a disjunction of literals. Then ϕ is a *disjunctive landmark* for P if every solution for P produces a state

```
RPG-Landmarks($\Sigma, s, g$)
    $Queue \leftarrow \langle$all literals in $g\rangle$; $Examined \leftarrow \varnothing$
    while $Queue \neq \langle\rangle$ do
        $\phi \leftarrow$ pop($Queue$)
1       if $\phi \notin Examined$ and $s \not\models \phi$ then
            // Step 1: get an action landmark
2           $R \leftarrow \{a \mid$ eff($a$) includes a literal in $\phi\}$
            // Step 2: get a smaller action landmark
3           $\hat{s}_0 \leftarrow s$
            for $k \leftarrow 1$ by $1$ until $\hat{s}_k = \hat{s}_{k-1}$ do
                $A_k \leftarrow \{a \in A \setminus R \mid a$ is r-applicable in $\hat{s}_{k-1}\}$
4               $\hat{s}_k \leftarrow \gamma^+(\hat{s}_{k-1}, A_k)$
5           $N \leftarrow \{a \in R \mid a$ is r-applicable in $\hat{s}_k\}$
            if $N = \varnothing$ then return failure
            // Step 3: get disjunctive landmarks
6           $\Phi \leftarrow \{$every disjunction $\phi'$ of $\leq 4$ preconditions of actions in $N$
              such that every $a \in N$ has at least one precondition in $\phi'\}$
7           append to $Queue$ every $\phi' \in \Phi$ that isn't subsumed by another
              member of $\Phi$
            add $\phi$ to $Examined$
    return $Examined$
```

Algorithm 3.9. RPG-Landmarks, an algorithm to find disjunctive landmarks by using relaxed planning graphs.

in which ϕ is true. The problem of deciding whether an arbitrary ϕ is a disjunctive landmark is PSPACE-complete, but that is a worst-case result. There are several polynomial-time algorithms for discovering some (though not necessarily all) of a problem's disjunctive landmarks.

Algorithm 3.9, RPG-Landmarks, uses relaxed planning graphs to look for disjunctive landmarks, starting with the goals in g and going backward toward s_0. It maintains a queue of landmarks to examine. In each iteration of its **while** loop, it takes a landmark ϕ from the queue and performs the following steps to look for other landmarks that precede ϕ:

Step 1: get an action landmark. At Line 2, all plans that achieve ϕ from s must contain one or more actions in R. Thus R is an *action landmark* for ϕ.

Step 2: get a smaller action landmark. When the relaxed-planning-graph computation at lines 3–4 finishes, $\hat{s}_k$ includes every atom that is achievable without using R. Thus at Line 5, $N \subseteq R$ includes every action in R that can be made executable without using R. The other actions in R cannot be made executable without using N, so N is another action landmark. If $N = \varnothing$ then (Σ, s, g) is unsolvable; otherwise RPG-Landmarks continues to the next step.

Step 3: get disjunctive landmarks. If we take one precondition from each action in N, then the disjunction of these preconditions is a landmark that precedes ϕ. Rather than computing all such landmarks (which would cause a combinatorial explosion), Line 6 lets Φ be the set of all such landmarks that contain 4 or fewer literals. Line 7 adds to the queue the ones that the others don't subsume.

After its queue is exhausted, RPG-Landmarks returns the landmarks that it found. Although it is more complicated than HFF, its running time is still polynomial. The *RPG Landmark heuristic* is

$$h^{\mathrm{RL}}(s) = \text{the number of landmarks returned by RPG-Landmarks}(\Sigma, s, g). \tag{3.8}$$

This is a relatively simple landmark heuristic, and there are many ways to improve it (see Section 3.6.5). Furthermore, planning algorithms can be made to perform better by considering the order in which to try to achieve landmarks. For example, a solution plan will need to achieve every landmark ϕ' in Line 7 before achieving ϕ.

Example 3.5. As before, consider the planning problem (Σ, s_0, g) in Example 3.2.

To compute $h^{\mathrm{RL}}(s_1)$, we call RPG-Landmarks(Σ, s_1, g). Every solution for (Σ, s_1, g) must include a state in which cargo(r1) = c1, and this is the only landmark that RPG-Landmarks will return. It will not return loc(r1) = d1, which is already true in s_1. Thus $h^{\mathrm{RL}}(s_1) = 1$.

For the planning problem (Σ, s_2, g), RPG-Landmarks will return two landmarks: cargo(s1) = c1 and loc(r1) = d1. Thus $h^{\mathrm{RL}}(s_2) = 2$. □

3.2.4 Max-Cost and Additive Cost Heuristics

The *max-cost* of a set of literals $g = \{l_1, \ldots, l_k\}$ is defined recursively as the largest max-cost of each literal l_i. Each l_i's max-cost is the minimum, over all actions a that can produce l_i, of a's cost plus the max-cost of pre(a). Here are the equations:

$$\begin{aligned}
\Delta^{\max}(s, g) &= \max_{l_i \in g} \Delta^{\max}(s, l_i);\\
\Delta^{\max}(s, l_i) &= \begin{cases} 0, & \text{if } l_i \in s,\\ \min\{\Delta^{\max}(s,a) \mid a \in A \text{ and } l_i \in \mathrm{eff}(a)\}, & \text{otherwise;}\end{cases}\\
\Delta^{\max}(s, a) &= \mathrm{cost}(a) + \Delta^{\max}(s, \mathrm{pre}(a)).
\end{aligned} \tag{3.9}$$

In a planning problem $P = (\Sigma, s_0, g)$, the *max-cost heuristic* is

$$h^{\max}(s) = \Delta^{\max}(s, g). \tag{3.10}$$

As shown in the following example, the computation of $h^{\max}$ can be visualized as an And/Or search going backward from g.

Example 3.6. In Example 3.2, suppose GBFS's heuristic function is $h^{\max}$. Figure 3.5 shows the computation of $h^{\max}(s_1) = 1$ and $h^{\max}(s_2) = 2$. Because $h^{\max}(s_1) < h^{\max}(s_2)$, GBFS will choose s_1. □

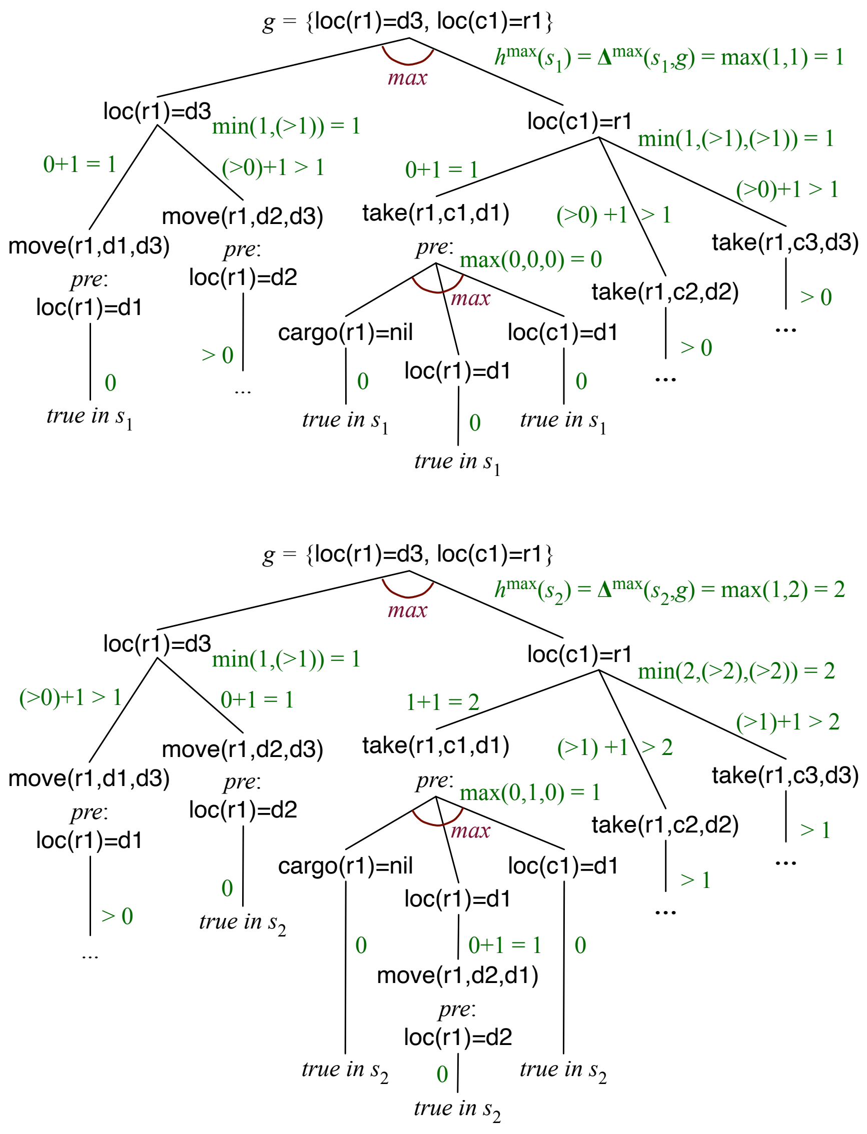


Figure 3.5. Computation of $h^{\max}(s_1, g)$ and $h^{\max}(s_2, g)$. The max and min operations in Equation 3.9 are shown as And-branches and Or-branches, respectively.

At the beginning of Section 3.2, we said that most heuristics are derived by relaxation. The $h^{\max}$ heuristic can be described as the cost of an optimal solution to a relaxed problem in which a goal (i.e., a set of literals such as g or the preconditions of an action) can be reached by achieving just one of the goal's literals, namely, the one that is the most expensive to achieve. It can also be described as a delete-relaxation heuristic and as a landmark heuristic (see Section 3.6.5).

Although $h^{\max}$ is admissible, it is not very informative. A closely related heuristic, the *additive cost* heuristic, is not admissible but generally works better in practice. It is similar to $h^{\max}$ but adds the costs of each set of literals rather than taking their

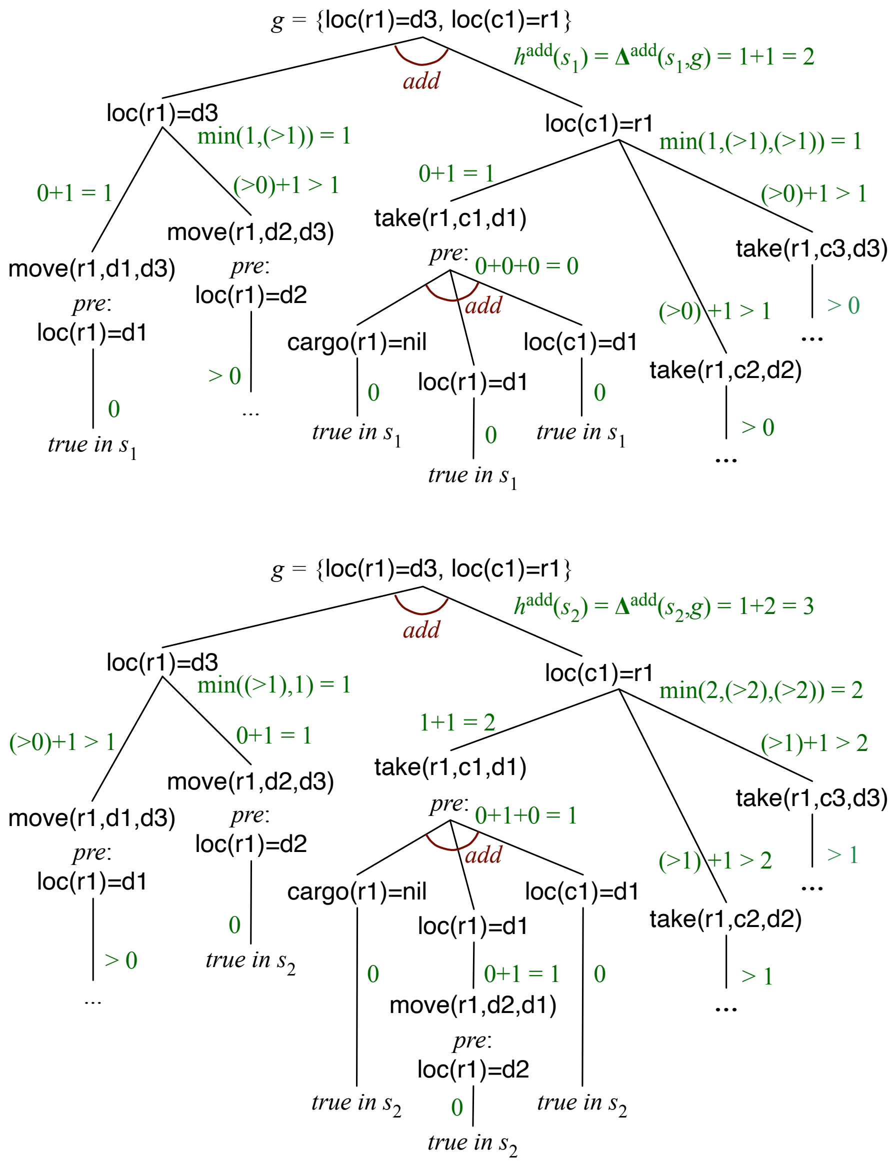


Figure 3.6. Computation of $h^{\text{add}}(s_1, g)$ and $h^{\text{add}}(s_2, g)$. The $\sum$ and min operations in Equations 3.12 and 3.13 are shown as And-branches and Or-branches, respectively.

maximum. It is defined as

$$h^{\text{add}}(s) = \Delta^{\text{add}}(s, g), \tag{3.11}$$

where

$$\Delta^{\text{add}}(s, g) = \sum_{l_i \in g} \Delta^{\text{add}}(s, l_i); \tag{3.12}$$

$$\Delta^{\text{add}}(s, l_i) = \begin{cases} 0, & \text{if } l_i \in s, \\ \min\{\Delta^{\text{add}}(s, a) \mid l_i \in \text{eff}(a)\}, & \text{otherwise;} \end{cases} \tag{3.13}$$

$$\Delta^{\text{add}}(s,a) = \text{cost}(a) + \Delta^{\text{add}}(s, \text{pre}(a)). \tag{3.14}$$

As shown in the following example, the computation of h^{add} can be visualized as an And/Or search nearly identical to the one for h^{max}.

Example 3.7. In Example 3.2, suppose GBFS's heuristic function is h^{add}. Figure 3.6 shows the computation of $h^{\text{add}}(s_1) = 2$ and $h^{\text{add}}(s_2) = 3$. Because $h^{\text{add}}(s_1) < h^{\text{add}}(s_2)$, GBFS will choose s_1.

To see that h^{add} is not admissible, notice that if a single action a could achieve both loc(r1)=d3 and loc(c1)=r1, then $h^{\text{add}}(g)$ would be higher than $h^*(g)$, because h^{add} would count a's cost twice. □

Both h^{max} and h^{add} have the same time complexity, which (like HFF) is polynomial in the total number of actions and ground atoms in the planning domain.

3.3 Backward Search

This section describes an algorithm that does a state-space search going backward from a goal g. To begin, we will need to define the conditions that make an action a useful as the last step of a plan to achieve g:

Definition 3.8. An action a is *consistent* with a set of literals g if it satisfies the first two of the following restrictions, and *relevant* for g if it satisfies all three of them:[7]

1. *a makes no condition in g false*: For every literal $x \neq v$ (or $x = v$) in g, eff(a) does not contain $x \leftarrow v$ (or respectively, $x \leftarrow v'$ for some $v' \neq v$).
2. *a does not require any condition in g to be false*: For every literal $x \neq v$ (or $x = v$) in g that is not affected by eff(a), pre(a) does not contain $x = v$ (or respectively, $x = v'$ for some $v' \neq v$).
3. *a makes at least one condition in g true*: For at least one literal $x = v$ (or $x \neq v$) in g, eff(a) contains $x \leftarrow v$ (or respectively, $x \leftarrow v'$ for some $v \neq v'$). □

We can now define $\gamma^{-1}(g,a)$ to be the conditions that must hold in every state s such that $\gamma(s,a) \models g$. If a is consistent with g, then

$$\begin{aligned}\gamma^{-1}(g,a) = \text{pre}(a) &\cup \{x = v \text{ in } g \mid \text{eff}(a) \text{ doesn't contain } x \leftarrow v\} \\ &\cup \{x \neq v \text{ in } g \mid \text{eff}(a) \text{ doesn't contain } x \leftarrow v' \text{ for any } v' \neq v\},\end{aligned} \tag{3.15}$$

and if a isn't consistent with g then $\gamma^{-1}(g,a)$ is undefined. By extension, we let

$$\gamma^{-1}(g,\pi) = \begin{cases} g, & \text{if } \pi \text{ is empty,} \\ \gamma^{-1}(\gamma^{-1}(g,\pi'),a), & \text{if } \pi = a\cdot\pi' \text{ and } a \text{ is consistent with } \gamma^{-1}(g,\pi'). \end{cases} \tag{3.16}$$

[7]When testing for relevance, backward-search algorithms sometimes omit one or both of the consistency restrictions. This simplifies the implementation, at the risk of misclassifying some actions as relevant that cannot be the last step of a plan to achieve g. For an example, see Line 2 of RPG-Landmarks.

As a special case, suppose $g = \varnothing$. In this case, Equation 3.15 reduces to $\gamma^{-1}(\varnothing, a) = \text{pre}(a)$, and Equation 3.16 gives the conditions that a state must satisfy for π to be applicable. Thus we define

$$\text{pre}(\pi) = \gamma^{-1}(\varnothing, \pi). \tag{3.17}$$

```
Backward-Search($\Sigma, s_0, g_0$)
1   $g \leftarrow g_0$; $\pi \leftarrow \langle\rangle$
    while $s \not\models g$ do
2       $A' \leftarrow \{a \mid a$ is relevant for $g\}$
        if $A' = \varnothing$ then return failure
3       nondeterministically choose $a \in A'$
4       $g \leftarrow \gamma^{-1}(g, a)$
        $\pi \leftarrow a \cdot \pi$
    return ($\pi$)
```

Algorithm 3.10. Backward-Search planning algorithm. In each loop iteration, π is a plan that can achieve g_0 from any state that satisfies g.

We now are ready to present Backward-Search. It is similar to Forward-Search except that it goes backward from the goal instead of forward from the initial state. We can incorporate cycle checking into it by adding the line

$$\textit{Solvable} \leftarrow \{g\}$$

after Line 1, and the following lines after Line 4:

```
if $g \in Solvable$ then return failure
$Solvable \leftarrow Solvable \cup \{g\}$
```

A more powerful form of cycle checking is to replace the preceding two lines with the following *subsumption* test:

```
if $\exists g' \in Solvable$ s.t. $g' \subseteq g$ then return failure
$Solvable \leftarrow Solvable \cup \{g\}$
```

Here, each $g' \in \textit{Solvable}$ represents a set of states from which π or one of its suffixes can reach g_0. If these "solvable" states include every state that $a \cdot \pi$ can solve, then it is useless to continue searching beyond $a \cdot \pi$. For any solution that ends with $a \cdot \pi$, another branch of the search space will contain a shorter solution that omits a.

Example 3.9. Suppose we augment Backward-Search to incorporate cycle checking and call it on the planning problem in Example 3.2. The first time through the loop,

$$\begin{aligned} g &\leftarrow \{\text{cargo(r1)} = \text{c1}, \text{loc(r1)} = \text{d3}\}, \\ A' &\leftarrow \{\text{move(r1, d1, d3)}, \text{move(r1, d2, d3)}, \text{take(r1, c1, d3)}\}. \end{aligned}$$

In Line 3, suppose Backward-Search chooses $\mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})$. Then

$$\begin{aligned} g &\leftarrow \gamma^{-1}(g, \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})) = \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}\};\\ \pi &\leftarrow \langle \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})\rangle;\\ \mathit{Solvable} &\leftarrow \{\{\mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}, \mathsf{loc}(\mathsf{r1}) = \mathsf{d3}\}, \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}\}\}. \end{aligned}$$

In the second loop iteration,

$$A' \leftarrow \{\mathsf{move}(\mathsf{r1}, \mathsf{d2}, \mathsf{d1}), \mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d1}), \mathsf{take}(\mathsf{r1}, \mathsf{c1}, \mathsf{d1})\}.$$

Let us consider two of the possible choices at Line 3:

1. If Backward-Search chooses $\mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d1})$, then

$$\begin{aligned} g &\leftarrow \gamma^{-1}(g, \mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d1})) = \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d3}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}\};\\ \pi &\leftarrow \langle \mathsf{move}(\mathsf{r1}, \mathsf{d3}, \mathsf{d1}), \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})\rangle;\\ g &\in \mathit{Solvable}, \text{ so Backward-Search returns failure.} \end{aligned}$$

2. If Backward-Search chooses $\mathsf{take}(\mathsf{r1}, \mathsf{c1}, \mathsf{d1})$, then

$$\begin{aligned} g &\leftarrow \gamma^{-1}(g, \mathsf{take}(\mathsf{r1}, \mathsf{c1}, \mathsf{d1})) = \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{nil}\};\\ \pi &\leftarrow \langle \mathsf{take}(\mathsf{r1}, \mathsf{c1}, \mathsf{d1}), \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})\rangle;\\ \mathit{Solvable} &\leftarrow \{\{\mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}, \mathsf{loc}(\mathsf{r1}) = \mathsf{d3}\}, \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{c1}\},\\ &\qquad \{\mathsf{loc}(\mathsf{r1}) = \mathsf{d1}, \mathsf{cargo}(\mathsf{r1}) = \mathsf{nil}\}\}. \end{aligned}$$

   If Backward-Search chooses $\mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})$ in the third loop iteration, then at the start of the fourth loop iteration it will return

$$\pi \leftarrow \langle \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3}), \mathsf{take}(\mathsf{r1}, \mathsf{c1}, \mathsf{d1}), \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d3})\rangle. \qquad \square$$

To choose among actions in A', Backward-Search can use many of the same heuristic functions described in Section 3.2, but with the following modification: rather than using them to estimate the cost of getting from the current state to the goal, what should be estimated is the cost of getting from s_0 to $\gamma^{-1}(g, a)$.

Often an action schema α may have multiple instances that are relevant for g, leading to a combinatorial explosion in the size of Backward-Search's search space. This problem can be alleviated by writing a Lifted-Backward-Search algorithm that leaves some of α's parameters uninstantiated. However, the details are complicated and we will omit them.[8]

[8]To find partially instantiated actions that satisfy a single atom in g, one could use a simple matching algorithm similar to Algorithm 5.6. However, Lifted-Backward-Search needs to find partially-instantiated actions that satisfy one *or more* literals in g, which requires a unification algorithm. See [967, Section 9.2.1] for an explanation and [535] for an algorithm.

3.4 Plan-Space Planning

Plan-space planning has some similarities to backward search, but it formulates planning as a constraint satisfaction problem and uses constraint-satisfaction techniques to produce solutions that are more flexible than linear sequences of ground actions. For example, it can produce plans in which the actions are partially ordered, along with a guarantee that every total ordering that is compatible with this partial ordering will be a solution plan.

Such flexibility allows some of the ordering decisions to be postponed until the plan is being executed, at which time the actor may have a better idea about which ordering will work best. Furthermore, the techniques used for plan-space planning are a first step toward planning concurrent execution of temporal actions, a topic that we will develop further in Part VI.

3.4.1 Definitions

Plan-space planning involves making repeated modifications to a plan in which the actions are both partially ordered and partially instantiated, as defined below.

A *partially ordered plan* is a triple

$$\pi = (V, E, act), \tag{3.18}$$

where V and E are the nodes and edges of an acyclic digraph and *act* is a function that maps each node $v \in V$ into an action, $act(v)$, so that actions may occur more than once. The edges in E represent ordering constraints on the nodes, and we define $v \prec v'$ if $v \neq v'$ and (V, E) contains a path from v to v'. Thus π represents a partially ordered multiset in which *act* is the labeling function.

If $v_1, \ldots, v_n$ is any ordering of the nodes such that $i < j$ whenever $v_i \prec v_j$, then the plan

$$\pi' = \langle act(v_1), \ldots, act(v_n) \rangle \tag{3.19}$$

is a *total ordering* of π. A *partially ordered solution* for a planning problem P is a partially ordered plan π such that every total ordering of π is a solution for P.

Definition 3.10. A *partial plan* is a 4-tuple

$$\pi = (V, E, act, C), \tag{3.20}$$

where (V, E, act) is a partially ordered plan as in Equation 3.18, except that for each $v \in V$, the action $act(v)$ may be unground. C is a set of constraints, each of which is one of the following:

- An *inequality constraint* is an expression $z \neq z'$, where z is an object variable, and z' is either an object variable or a constant.
- A *causal link* is an expression of the form

$$v_1 \overset{x=b}{\dashrightarrow} v_2, \tag{3.21}$$

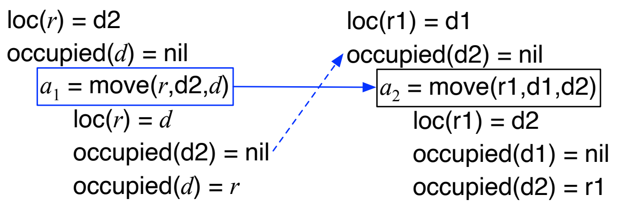


Figure 3.7. A partial plan that contains a causal link. Each action's preconditions and effects are shown near its upper left corner and lower right corner, respectively.

where v_1 and v_2 are nodes, $v_1 \prec v_2$, the effects of $act(v_1)$ include $x \leftarrow b$, and the preconditions of $act(v_2)$ include either $x = b$ or a literal $x \neq b'$ for some $b' \neq b$. The causal link's purpose is to assert that $act(v_1)$ is the action that establishes the given precondition of $act(v_2)$. If a node v_3 such that $v_1 \prec v_3 \prec v_2$ has an effect $x \leftarrow t$ for some t, then v_3 *violates* the causal link, even if $t = b$.[9] □

Example 3.11. Let Σ be a planning domain in which *Objects* = *Robots* ∪ *Docks*, where *Robots* = {r1, r2} and *Docks* = {d1, d2, d3}. There are no rigid relations, and one action schema, where $r \in$ *Robots* and $d, d' \in$ *Docks*:

$$\begin{array}{l} \mathsf{move}(r, d, d') \\ \quad \text{pre: } \mathsf{loc}(r) = d,\ \mathsf{occupied}(d') = \mathsf{nil} \\ \quad \text{eff: } \mathsf{loc}(r) \leftarrow d',\ \mathsf{occupied}(d') = r \end{array}$$

Let $\pi = (V, E, act, C)$ be the following partial plan:

$$\begin{aligned} V &= \{v_1, v_2\}, \\ E &= \{(v_1, v_2)\}, \\ act(v_1) &= \mathsf{move}(r, \mathsf{d2}, d), \\ act(v_2) &= \mathsf{move}(\mathsf{r1}, \mathsf{d1}, \mathsf{d2}, \\ C &= \{v_1 \xrightarrow{\mathsf{occupied}(\mathsf{d2}) = \mathsf{nil}} v_2\}. \end{aligned}$$

The plan is shown in Figure 3.7, with the edge in E represented by a solid arrow and the causal link represented by a dashed arrow. □

A partial plan $\pi = (V, E, act, C)$ is *inconsistent* in each of the following situations: if (V, E) contains a cycle, if C contains a self-contradictory inequality constraint (e.g., $y \neq y$), if there is a violated causal link, or if an action $act(v)$ has an illegal argument. Otherwise π is *consistent*.

Definition 3.12. A *partial solution* for a planning problem $P = (\Sigma, s_0, g)$ is a partial plan $\pi = (V, E, act, \varnothing)$ in which s_0 and g are represented by *dummy actions* a_0 and a_g that are not instances of action schemas in $\mathcal{A}$. Their sole purpose is to represent s_0 and g in a way that is easy for PSP to work with. More specifically,

[9]The reason for calling this a violation when $t = b$ is to ensure PSP (which will be defined in the next section) performs a *systematic* search [770, 572], that is, it does not generate the same partial plan several times in different parts of the search space. This reduces the size of PSP's search space.

- V contains nodes v_0 and v_g such that $act(v_0) = a_0$ and $act(v_g) = a_g$;
- a_0 has $\text{pre}(a_0) = \varnothing$ and $\text{eff}(a_0) = s_0$;
- a_g has $\text{pre}(a_g) = g$ and $\text{eff}(a_g) = \varnothing$;
- the ordering constraints in E must be such that $v_0 \prec v \prec v_g$ for every node $v \notin \{v_0, v_g\}$. □

```
PSP($\Sigma, \pi$)
    while $Flaws(\pi) \neq \varnothing$ do
1       arbitrarily select $f \in Flaws(\pi)$
        $R \leftarrow$ {all feasible resolvers for $f$}
        if $R = \varnothing$ then return failure
2       nondeterministically choose $\rho \in R$
3       modify $\pi$ by applying $\rho$ to it
    return ($\pi$)
```

Algorithm 3.11. PSP, a plan-space planning algorithm. If the partial plan π represents a solvable planning problem in the planning domain Σ, then at least one of PSP's nondeterministic traces will return a solution plan.

3.4.2 Planning Algorithm

Algorithm 3.11, PSP, takes as input a planning domain Σ and a partial solution π that represents a planning problem $P = (\Sigma, s_0, g)$. To try to solve P, PSP repeatedly looks for *flaws* in π and applies *resolvers* to remove the flaws.

In PSP, $Flaws(\pi)$ is the set of all flaws in π. There are two kinds of flaws: open goals and threats. These are described next, along with their resolvers.

Open goals. If a node $v \in V$ has a precondition $p \in \text{pre}(act(v))$ for which there is no causal link, then p is an *open goal*. There are two kinds of resolvers for this flaw:

- *Use an action already in* π. Suppose π contains a node v' such that $v \not\prec v'$ and $act(v')$ has an effect e that can be *unified* with p, that is, e and p can be made syntactically identical by instantiating some of the object variables in π. Then the flaw can be resolved by unifying e and p, adding a causal link $v' \overset{e'}{\dashrightarrow} v$ in which e' is the unified expression, and adding (v', v) to E so that $v' \prec v$.
- *Use a new action.* Let α be an action schema and a be a *standardization* of α, that is, a copy of α in which object variables are renamed to avoid name conflicts with the object variables in π.[10] If $\text{eff}(a)$ includes an effect e such that p is an instance of e, then the flaw can be resolved by adding to π a new node v' with $act(v') = a$, instantiating variables of a to make e match p, adding a causal link $v' \overset{p}{\dashrightarrow} v$, and adding edges (v_0, v') and (v', v) to E so that $v_0 \prec v' \prec v$.

[10]This is analogous to standardization in logical inference (see [967, Section 9.2.1]).

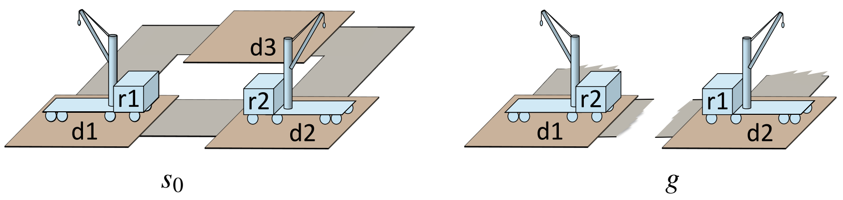


Figure 3.8. Initial state and goal for Example 3.13.

Threats. Let $v_1 \overset{x=b}{\dashrightarrow} v_2$ be any causal link in π, and $v \in V$ be any node such that $v \nprec v_1$ and $v_2 \nprec v$ (that is, v may come between v_1 and v_2). Suppose $act(v)$ has an effect $x' \leftarrow t$ such that the state variable x' can be unified with x. Then v is a *threat* to the causal link. There are three kinds of resolvers for such a threat:

- Make $v \prec v_1$ by adding (v, v_1) to E.
- Make $v_2 \prec v$ by adding (v_2, v) to E.
- Add to C an inequality constraint that prevents x and x' from unifying.

Example 3.13. Let Σ be the planning domain in Example 3.11, and consider the planning problem $P = (\Sigma, s_0, g)$, where

$$\begin{aligned} s_0 = \{&\mathsf{loc(r1) = d1, loc(r2) = d2,}\\ &\mathsf{occupied(d1) = r1, occupied(d2) = r2, occupied(d3) = nil}\};\\ g = \{&\mathsf{loc(r1) = d2, loc(r2) = d1}\}. \end{aligned}$$

Figure 3.8 shows s_0 and g, and Figure 3.9 shows the initial partial plan. Figures 3.10–3.13 show some snapshots of one of PSP's nondeterministic execution traces. Solid arrows represent edges in E, dashed arrows represent causal links, and thick dot-dashed arrows represent threats. □

Like Forward-Search and Backward-Search, PSP is sound and complete; but unlike them, it may often have infinite paths in its search space. Thus it is not guaranteed to terminate on unsolvable problems.

3.4.3 Search Heuristics

Several of the choices that PSP must make during its search are very similar to the choices that a backtracking search algorithm makes in order to solve constraint-satisfaction problems (CSPs); for example, see [967]. Consequently, some of the heuristics to guide CSP algorithms can be translated into heuristics to guide PSP:

- Because all of the flaws must eventually be resolved, flaw selection in Line 1 of PSP is not a nondeterministic choice. However, the order in which PSP selects the flaws can affect the size of the search space generated by PSP's nondeterministic choices in Line 2. Flaw selection is analogous to variable ordering in CSPs, and the Minimum Remaining Values heuristic for CSPs (choose the variable with the fewest remaining values) is analogous to a PSP heuristic called *Fewest Alternatives First*: select the flaw with the fewest resolvers.

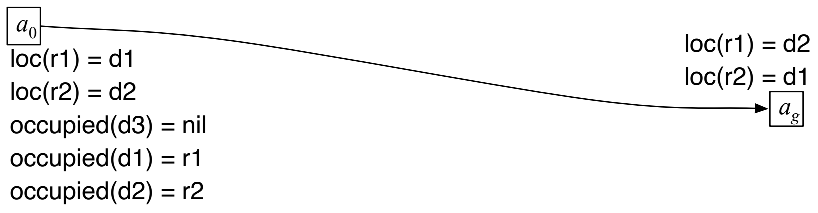


Figure 3.9. The initial partial plan. The dummy actions a_0 and a_g represent s_0 and g. There are two open-goal flaws: a_g's preconditions loc(r1) = d2 and loc(r2) = d1.

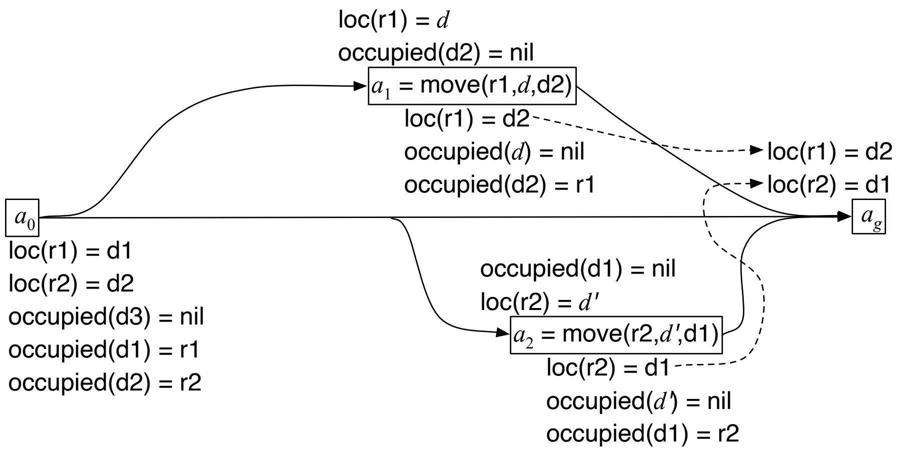


Figure 3.10. Resolving a_g's open-goal flaws. For loc(r1) = d2, PSP adds action a_1 and a causal link. For loc(r2) = d1, PSP adds action a_2 and another causal link. This adds four new open-goal flaws: the preconditions of a_1 and a_2.

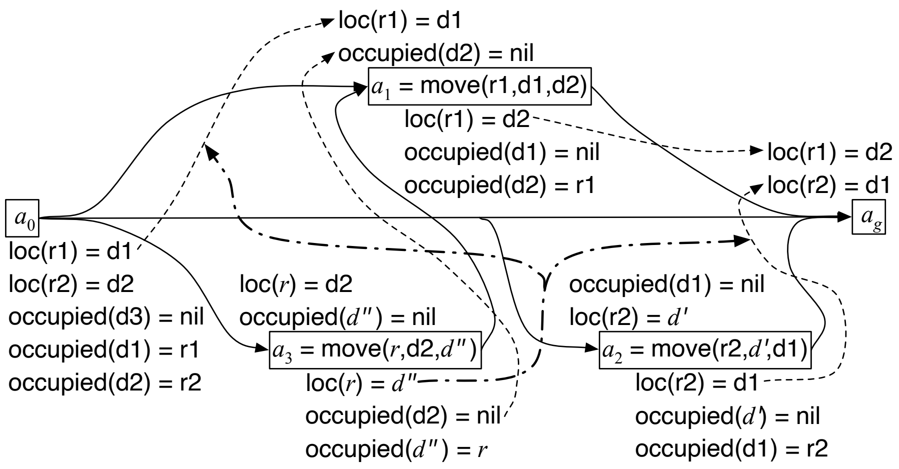


Figure 3.11. Resolving a_1's open-goal flaws. For loc(r1) = d, PSP instantiates d to d1 and adds a causal link from a_0. For occupied(d2) = nil, PSP adds action a_3 and a causal link. The new action causes two threats, shown as dashed-dotted lines.

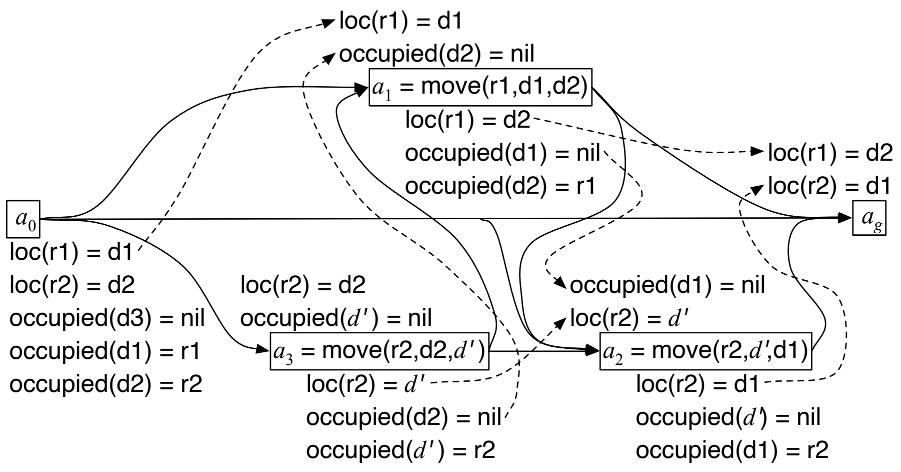


Figure 3.12. Resolving a_2's open-goal flaws. For occupied(d1) = nil, PSP adds a causal link from a_1. For loc(r2) = d', PSP adds a causal link from a_3, where it instantiates r to r2 and d'' to d'. These changes also resolve the two threats.

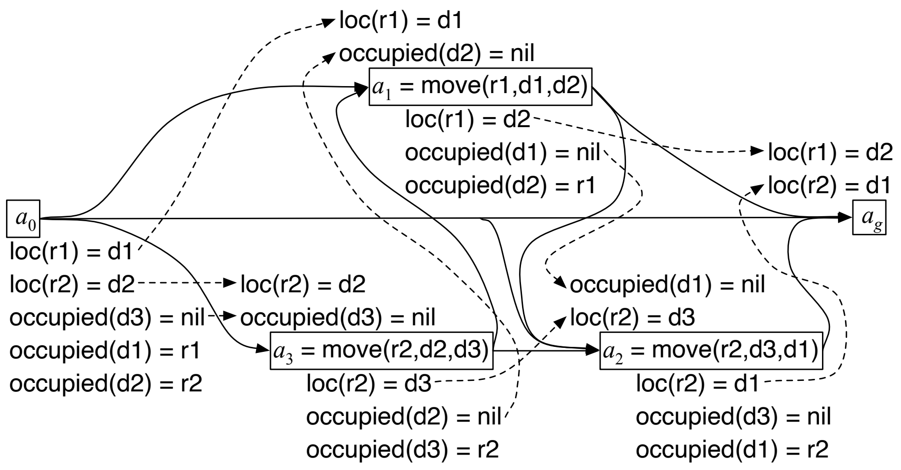


Figure 3.13. Resolving a_3's open-goal flaws. For loc(r2) = d2, PSP adds a causal link from a_0. For occupied(d3) = nil, PSP instantiates d' to d3 and adds a causal link from a_0. There are no further flaws, so this is a partially-ordered solution.

- Resolver selection in Line 2 of PSP is analogous to value ordering in CSPs. The Least Constraining Value heuristic for CSPs is to choose the value that rules out the fewest values for the other variables. One might want to translate this into a "least-constraining resolver" heuristic for PSP: choose the resolver that rules out the fewest resolvers for the other flaws. Unfortunately, this ignores an important difference between plan-space planning and CSPs.

  In a CSP, the number of variables is ordinarily fixed in advance, the search space is finite, and all solutions are at the same depth. In PSP, the least-constraining resolver may introduce a new action. This may occur arbitrarily many times, and each occurrence is analogous to introducing new variables (and new constraints) into a CSP: it increases the size of the search space.

One way to fix this problem might be to look first for resolvers that do not introduce open goals—and if there are several such resolvers, *then* to choose the one that rules out the fewest resolvers for the other flaws.

Although these heuristics can help speed PSP's search, implementations of PSP tend to run much more slowly than the fastest state-space planners. Generally the latter are GBFS algorithms that are guided by heuristics like the ones in Section 3.2, and there are several impediments to developing an analogous version of PSP. Because plan spaces have no explicit states, the heuristics in Section 3.2 are not directly applicable, nor is it clear how to develop similar plan-space heuristics. Even if such heuristics were available, a depth-first implementation of PSP would be problematic because plan spaces generally are infinite. Thus for solving classical planning problems such as the ones in the International Planning Competitions, most automated-planning researchers have abandoned PSP in favor of forward-search algorithms.

On the other hand, the hybrid-planning algorithms in Sections 5.3 and 17.2 are based on PSP. They are much easier to understand if one first understands PSP.

3.5 Repairing Plans

Plan repair can provide advantages over generating new plans from scratch, both in terms of the runtime needed for planning and the plan's *stability*, that is, the amount of the original plan π that is retained in the repaired plan. As discussed in Section 2.6.4, plan stability may be important if the actor needs to coordinate with other actors that are depending on π, avoid wasting resources that were acquired for use later in π, or avoid making changes that may be confusing to human users.

```
Incremental-Repair($\Sigma, s, g, \pi$)
  while True do
1     $\pi' \leftarrow$ Lookahead($\Sigma, s, \gamma^{-1}(g, \pi)$)
      if $\pi' \neq$ failure then return $\pi' \cdot \pi$
      if $\pi = \langle\rangle$ then return failure
      $a \leftarrow$ pop($\pi$)
```

Algorithm 3.12. Incremental-Repair tries to retain π's largest possible suffix.

Incremental-Repair attempts to repair π in a way that retains the largest possible suffix of π. First it looks for a plan π' such that $\pi' \cdot \pi \models g$. If that succeeds, it returns $\pi' \cdot \pi$. Otherwise it removes the first action from π and tries again, proceeding in this manner until either it succeeds or none of π is left.

Line 1 of Incremental-Repair uses an extended definition of γ^{-1} (see Equation 3.15) that includes plans. If π is a plan, then $\gamma^{-1}(g, \pi)$ is the condition that a state s must satisfy to ensure that $\gamma(s,\pi) \models g$. More formally,

$$\gamma^{-1}(g,\pi) = \begin{cases} g, & \text{if } \pi = \langle\rangle, \\ \gamma^{-1}(\gamma^{-1}(g,a),\pi'), & \text{if } \pi = \pi' \cdot a \text{ for some } \pi' \text{ and } a, \\ \text{undefined}, & \text{otherwise.} \end{cases} \tag{3.22}$$

Incremental-Repair is a simple algorithm that provides no guarantee of finding the best way to repair π. Other approaches to plan repair are discussed in Section 3.6.9.

3.6 Discussion and Bibliographic Notes

3.6.1 Nondeterministic Algorithms

Many of the planning algorithms in this book will be presented as nondeterministic search algorithms, like the Forward-Search algorithm at the beginning of this chapter. Line 1 of Forward-Search corresponds to trying several members of R sequentially in a trial-and-error fashion. The command "**nondeterministically choose**" is an abstraction that lets us ignore the precise order in which those values are tried. This lets us discuss properties that are shared by a wide variety of algorithms that search the same space of partial solutions, even though those algorithms may visit different nodes of that space in different orders. Initially these were just called nondeterministic algorithms [366, 246], but this kind of nondeterminism later came to be called *angelic*, as distinguished from demonic and erratic nondeterminism [115].

3.6.2 Search Algorithms

Heuristic functions that estimated the distance to the goal were first developed in the mid-1960s [842, 721, 304], and the A* algorithm was developed a few years later [471, 472]. The ϵ-optimality result for A* is from [908]. A huge amount of subsequent work has been done on A* and other heuristic search algorithms. There are tutorial introductions to some of these algorithms [856, 967]. Our definition of problem relaxation in Section 3.2 is based on [877], which provides a comprehensive analysis of a large number of algorithms and techniques.

Branch-and-bound algorithms have been widely used in combinatorial optimization problems [813]. DFBB (Section 3.1.7) is the best-known version, but most forward-search algorithms (including, for example, A*) can be formulated as special cases of branch-and-bound [532, 831].

GBFS, which has been used in many classical planning algorithms [490, 392], was first introduced in [304] under a different name. Several enhancements to GBFS have been developed, such as combining it with local search [1187]. GBFS, and several other algorithms that find approximately optimal solutions, can be adapted to run in an "anytime mode" in which the algorithm does not stop at the first solution it finds, but instead continues to look for better and better solutions as time permits [468].

Prior to GBFS, the name "greedy best-first search" was used in [156] for a planning algorithm similar to the weighted A* algorithm in [908]. These algorithms use a heuristic function of the form $g + wh$ or $(1 - w)g + wh$, where $w > 1$ is a weight that gives h more influence than g. Such a weighting scheme has worked well in the LAMA planner [941].

Heuristic search algorithms can sometimes encounter "heuristic plateaus" that all look the same from the point of view of the heuristic function [373, 507]. To escape such plateaus, the well-known FF algorithm does a breadth-first search [509]. A more recent technique is a width-k search, which prunes all states except those in which at least k literals have become true for the first time along the current path [722].

In game-tree search programs for games such as chess and checkers, the acting procedure is similar to Run-Lookahead with the *Lookahead* subroutine being like a time-limited version of depth-first iterative deepening (Section 3.1.8) and the depth-first search being a variant of the well-known alpha-beta algorithm [619, 856, 967].

IDA* [640] and other iterative-deepening algorithms are a special case of *node-regeneration* algorithms that retract nodes to save space and regenerate them later if they need to examine them again. There are several other such algorithms [642, 415].

3.6.3 Planning Graphs

A *planning graph* is similar to HFF's relaxed planning graphs (see Figures 3.3 and 3.4), but it also includes various *mutex* (i.e., mutual exclusion) conditions: for example, two actions are mutex if they change the same state variable to different values. Rather than including all r-applicable actions, each A_k only includes the ones whose preconditions are not mutex in $\hat{s}_k$. A good tutorial account of this appears in [1162].

Planning graphs were first used in the Graphplan algorithm [146], which does an iterative-deepening search to generate successively larger r-states. For each r-state $\hat{s}_k$ such that the atoms of g are non-mutex in $\hat{s}_k$, Graphplan does a backward search to look for a relaxed solution π such that the actions in each $\hat{a}_i$ are non-mutex. Such a π is often called a *parallel plan* or *layered plan*, and it is a partially ordered solution (although not necessarily an optimal one). In any solvable planning problem, a sufficiently large planning graph will contain a solution, hence Graphplan is complete. Furthermore, because Graphplan's backward search is restricted to the planning graph, it usually can solve classical planning problems much faster than planners based on Backward-Search or PSP [1162].

Graphplan inspired much follow-up research on planning-graph techniques. Some of them extend planning graphs in various nonclassical directions, such as conformant planning [1034], sensing [1165], temporal planning [1035, 400, 733], resources [624, 625, 1049], probabilities [145], soft constraints [791], and distributed planning [545]. Others have combined planning graphs with other techniques for use on classical-planning problems. The STAN planner [732] uses a combination of efficient planning-graph implementation and domain analysis. LPG [399] does a stochastic local search on a network of the actions in the planning graph.

3.6.4 Translating Planning Problems into Other Problems

The BlackBox planner [591] can translate a planning problem or a planning graph into a satisfiability problem and search for a solution using a satisfiability solver. The basic idea is, for $n = 1, 2 \ldots,$ to take the problem of finding a plan of length n, rewrite it as a satisfiability formula f_n, and try to solve f_n. If the planning problem is solvable, then f_n will be solvable for sufficiently large n. Subsequent work on planning as satisfiability has included new translation algorithms [529, 949, 8], and planning-specific heuristics for variable selection in satisfiability problems [948].

There are related approaches that translate planning problems into constraint-satisfaction problems [298, 91, 92] or integer-programming problems [1113]; see [829] for an overview.

3.6.5 Heuristic Functions

The h^{add} and h^{max} heuristics in Section 3.2.4 were first used in the HSP planning system [156]. They were highly influential because they disproved a long-held assumption that good heuristic functions needed to be domain-specific. HSP performed excellently in the 1998 planning competition, the first of a planning-competition series[11] that has sparked much research on domain-independent planning heuristics.

Most domain-independent heuristics can be classified roughly as delete-relaxation heuristics, landmark heuristics, critical-path heuristics, and abstraction heuristics [492]. The next several paragraphs discuss each of these. There also are heuristic functions that combine multiple heuristic estimates [957].

Delete-Relaxation Heuristics. Delete-relaxation and the h^+ and h^{FF} heuristics (see Section 3.2.1) were pioneered primarily by Hoffmann [509], and the name h^{FF} derives from its use in the FF planning system [507]. However, instead of using h^{FF} in the way that we described, FF did something closer to the improvements described at the end of Section 3.2.2. Delete-relaxation can also be used to describe the h^{add} and h^{max} heuristics: h^{max} is the optimal parallel solution (see Section 3.6.3) for the delete-relaxed problem [493, 133].

The *causal-graph* heuristic [490] involves analyzing the planning domain's causal structure using a directed graph in which the nodes are the planning domain's state variables, and the edges represent dependencies among the state variables. Although it is not immediately obvious that this is a delete-relaxation heuristic, there is a delete-relaxation heuristic that includes it and h^{add} as special cases [493].

Landmark Heuristics. The early work on landmarks [913] was hugely influential, inspiring a great deal of additional work on the subject. The landmark heuristic that we described in Section 3.2.3 is relatively simple, and there are many ways to improve it. The problems of determining whether a fact is a landmark, or whether one landmark must precede another, are PSPACE-complete [510]. However, there are several polynomial-time criteria that are sufficient (but not necessary) to guarantee that a fact is a landmark or that one landmark must proceed another. Some of the better-known approaches involve relaxed planning graphs [510], domain transition graphs [942, 941], hitting sets [165], and cyclic dependencies [189].

[11] See https://www.icaps-conference.org/competitions/.

The $h^{\max}$ heuristic can also be described as a landmark heuristic [492]. An enhanced version, $h^{\text{LM-Cut}}$, is still admissible and gives close approximations to h^{+} [492]. It has been generalized to planning problems in which actions have conditional effects [958]. Landmark heuristics have also been developed for temporal [587] and numeric [983] planning problems.

Critical-Path Heuristics. There is a set $\{h^m \mid m = 1, 2, \ldots\}$ of admissible heuristic functions based loosely on critical paths (an important concept in project scheduling). Each h^m approximates the cost of achieving a goal g by the cost of achieving the most costly subset of size m [476, 478]. The computation is exponential in m, but runs in polynomial time for any fixed m.

Abstraction Heuristics. An *abstraction* of a planning domain Σ is a γ-preserving homomorphism from Σ onto a smaller planning domain Σ'. For each planning problem $P = (\Sigma, s_0, g)$, this defines a corresponding abstraction $P' = (\Sigma', s'_0, g')$. If c^* is the cost of an optimal solution to a planning problem, then $c^*(P') \leq c^*(P)$. If Σ' is simple enough that we can compute $c^*(P')$ for every planning problem P' in Σ', then the function $h(s) = c^*(\Sigma', s', g')$ is an admissible heuristic for P.

The best-known such abstraction is *pattern database* abstraction [263, 319]. The *pattern* is a subset X' of the state variables in Σ, and the mapping from Σ to Σ' is done by removing all literals that have state variables not in X'. The pattern database is a table that gives $c^*(P')$ for every planning problem P' in Σ'. There are algorithms to decide what to include in X' [479, 495], but unfortunately the size of the pattern database and the cost of computing each entry both grow exponentially with X'. This can be alleviated [320, 79] using symbolic representation techniques such as BDDs (Section 12.3.4), but X' still needs to be kept small [496]. The database provides no information about variables not in X', so this limits the informedness of h.

Awareness of this limitation has led to research on other criteria for aggregating sets of states in Σ into individual states in Σ', including merge-and-shrink abstraction [494, 496] and structural-pattern abstraction [589], as well as ways to improve the heuristic's informedness by composing several different abstractions [588, 496, 994].

3.6.6 Plan-Space Planning

The two earliest plan-space planners, NOAH [969] and NONLIN [1079], combined plan-space search with HTN task refinement (see Chapter 5). Plan-space planning was initially called *nonlinear* planning, reflecting some debate over whether "linear" planning referred to the structure of the planner's current set of actions (a sequence instead of a partial order) or to its search strategy that addresses one goal after the previous one has been completely solved.

The SNLP algorithm [770] introduced the concept of *systematic* search, in which a plan-space planner generates each partial plan at most once [572]. The footnote at the end of Definition 3.10 discusses systematic search in PSP. The UCPOP planner [883, 87, 1163] extended SNLP to handle some extensions to the classical domain representation, including conditional effects and universally quantified effects [879, 880]. Several other extensions have also been studied, such as incomplete information and sensing actions [886, 334, 429] and some kinds of extended goals [1164, 77].

Work on planning performance in plan-space planning has included studies of search control and pruning [397], commitment strategies [793, 794, 1213], state space versus plan space [1123], and domain features [618].

The general formulation of domain-independent planning in [578, 573] takes into account most of the preceding issues.

3.6.7 Generalized Domain Models

In Section 2.7.2 we discussed the possibility of generalizing the state-variable representation in Section 2.3 to allow actions to do arbitrary computations on states represented as arbitrary data structures. With such modifications, the forward-search algorithms in Section 3.1 will still work correctly [856, 654, 521], but they will not be able to use the domain-independent heuristic functions in Section 3.2, because those heuristics work by manipulating the syntactic elements of state-variable and classical representations. Instead, domain-specific heuristic functions will be needed.

One way to generalize the action model while still allowing domain-independent heuristics is to write each action as a combination of two parts—a "classical" part that uses a classical or state-variable representation and a "nonclassical" part that uses some other kind of representation—and write separate algorithms to reason about the classical and nonclassical parts. This approach has been used to combine classical planning and integer programming [480]. There is also a "planning modulo theories" approach [447] that was inspired by prior work on SAT modulo theories [852, 88].

For a planning system to work well, its domain and problem descriptions may need to be carefully engineered and fine-tuned for particular domains and problems. This can require an expert understanding of both the domain and the planning language [481]. An ongoing series of workshops focuses on ways to alleviate the task of knowledge engineering for planning. The 33rd such workshop was in 2023.[12]

3.6.8 Planning with Abstraction

In the AI planning literature, *planning with abstraction* usually has meant a relaxation process in which an *abstract* planning problem $P' = (\Sigma', s_0', g')$ is formed from a classical planning problem $P = (\Sigma, s_0, g)$ by removing some atoms and the literals that contain them [969, 617, 1199, 424]. If a planner finds a solution $\pi' = \langle a_1', \ldots, a_n' \rangle$ for P', then for each i, let a_i be the action whose abstraction is a_i'. Then we can constrain the search for a solution to P by treating $\text{pre}(a_1'), \ldots, \text{pre}(a_n')$ like landmarks:

$$\begin{aligned}
&\text{find a solution } \pi_0 \text{ for } P_0 = (\Sigma, s_0, \text{pre}(a_1)), && \\
&\text{find a solution } \pi_1 \text{ for } P_1 = (\Sigma, s_1, \text{pre}(a_2)), && \text{where } s_1 = \gamma(s_0, \pi_0), \\
&\qquad\ldots, && \\
&\text{find a solution } \pi_{n-1} \text{ for } P_{n-1} = (\Sigma, s_{n-1}, \text{pre}(a_n)), && \text{where } s_{n-1} = \gamma(s_{n-2}, \pi_{n-2}), \\
&\text{find a solution } \pi_n \text{ for } P_n = (\Sigma, s_n, g), && \text{where } s_n = \gamma(s_{n-1}, \pi_{n-1}).
\end{aligned}$$

[12]See https://icaps23.icaps-conference.org/program/workshops/keps/.

If a condition called the *downward refinement property* [68] holds, then $\pi_0, \dots, \pi_n$ will be guaranteed to exist, and their concatenation will be a solution for P. Planning with abstraction typically is done at multiple levels: use an abstraction P'' to constrain the search for solving P'; use an abstraction P''' to constrain the search for solving P''; and so forth. Such abstraction hierarchies have been extensively studied [85, 617, 1199].

In the abstract planning problem P', each state or action represents an equivalence class of states or actions in P. These equivalence classes were induced by the removal of atoms, but there are other ways to create equivalence classes with analogous properties and use them for planning with abstraction [754, 755].

Often the downward refinement property is not satisfied, and in such cases planning with abstraction is not guaranteed to work. However, abstracted planning problems can also be used to provide heuristic functions to guide the search for a solution to the unabstracted problem (see the *abstraction heuristics* part of Section 3.6.5).

3.6.9 Plan Repair

The term "plan stability" was introduced in [371], which used a modified version of the LPG planner [400] to show that plan repair could produce corrected plans more quickly and with fewer revisions than replanning from scratch. The term "minimal perturbation" was used synonymously in [266], which pointed out the importance of commitments to other agents.

There has been much classical planning research on the problem of *plan adaptation*, that is, taking a solution for one problem and modifying it to get a solution for another [465, 836, 63, 400, 820]. Because plan repair is a special case of plan adaptation, most of these algorithms can be used for plan repair. Fewer classical-planning works have focused on plan repair *per se*; one exception is the POPR plan-repair algorithm for plan-space plans [1114].

Work has also been done on plan repair in non-classical domains. There are domain-specific algorithms for a variety of domains [971, 1027, 463, 825, 706], and Section 2.6.4 will discuss plan-repair algorithms for HTN planning.

3.7 Exercises

3.1. Prove that in any solvable classical planning problem, at least one execution trace of Forward-Search will return a shortest solution. Do the same for Backward-Search.

3.2. Under what conditions will GBFS switch to a different path if its current path is not a dead end?

3.3. In the blocks-world planning problem in Exercise 2.4, let b be any block, and suppose its current location is $\mathsf{loc}(b) = l$ for some l. We will say that b *needs to be moved* if either the goal formula includes an atom $\mathsf{loc}(b) = l'$ such that $l' \neq l$, or there is a block below b that needs to be moved. Consider the heuristic function $h(s) = nm(s) - om(s)$, where

$nm(s) =$ the number of blocks that need to be moved;

$$s_0 = \{\text{loc(a)} = \text{b}, \text{loc(b)} = \text{table}, \text{loc(c)} = \text{table}, \text{clear(a)} = \text{T}, \text{clear(b)} = \text{F}, \text{clear(c)} = \text{T}\},$$
$$g = \{\text{loc(a)} = \text{b}, \text{loc(b)} = \text{c}\}$$

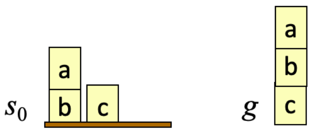


Figure 3.14. Initial state and goal for the planning problem in Exercise 3.3.

$om(s)$ = the number of blocks that are at most two moves away from a location where they won't need to be moved.

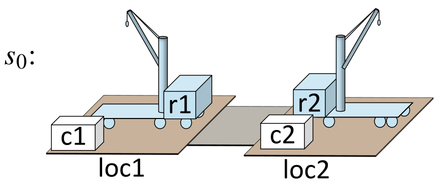
Using h, suppose we run GBFS on the planning problem in Figure 3.14. Draw the search tree that GBFS will produce. For each node in the tree, just draw the state, rather than writing it mathematically. Next to each state, write its h value.

take(r, c, l)
  pre: loc$(r) = l$, pos$(c) = l$,
    cargo(r) = nil
  eff: cargo$(r) \leftarrow c$, pos$(c) \leftarrow r$

put(r, c, l)
  pre: loc$(r) = l$, pos$(c) = r$
  eff: cargo$(r) \leftarrow$ nil, pos$(c) \leftarrow l$

move(r, l, m)
  pre: loc$(r) = l$
  eff: loc$(r) \leftarrow m$

where $r \in Robots$, $l, m \in Locations$,
  $c \in Containers$



$$s_0 = \{\text{loc(r1)} = \text{loc1}, \text{loc(r2)} = \text{loc2}, \text{cargo(r1)} = \text{nil}, \text{cargo(r2)} = \text{nil}, \text{pos(c1)} = \text{loc1}, \text{pos(c2)} = \text{loc2}\},$$

$$g = \{\text{pos(c1)} = \text{loc2}, \text{pos(c2)} = \text{loc2}\}$$

Figure 3.15. Planning problem for Exercise 3.4.

3.4. Figure 3.15 shows a planning problem involving two robots whose actions are controlled by a single actor. Unlike some of our previous examples, it is possible for both robots to occupy the same location.

(a) If we run Forward-Search on this problem, how many iterations will the shortest execution traces have, and what plans will they return? For one of them, give the sequence of states and actions in the execution trace.
(b) If we run Backward-Search on this problem, how many iterations will the shortest execution traces have, and what plans will they return? For one of them, give the sequence of goals and actions in the execution trace.
(c) Compute $h^{\text{FF}}(s_0)$.
(d) In HFF, suppose that instead of exiting the loop at the first value of k such that $\hat{s}_k$ r-satisfies g, we instead keep iterating the loop. At what value of k will $|\hat{s}_k|$ reach its maximum? At what value of k will $|A_k|$ reach its maximum?

(e) Compute $h^{\mathrm{RL}}(s_0)$.
(f) Compute $h^{\mathrm{add}}(s_0)$ and $h^{\max}(s_0)$.

3.5. Write pseudocode for the improved version of HFF that was described in the last paragraph of Section 3.2.2. Describe a similar improved version of RPG-Landmarks, and write pseudocode for it.

3.6. What might be an effective way to use h^{FF}, h^{RL}, h^{add}, and $h^{\max}$ with Backward-Search?

3.7. In the discussion of RPG-Landmarks, we remarked that a solution plan will need to achieve every landmark ϕ' in Line 7 before achieving the landmark ϕ.

(a) Modify RPG-Landmarks to make *Examined* a partially ordered set that uses the order in which the landmarks will need to be achieved.
(b) Describe a planning algorithm that uses the partially ordered set in part (a) to solve a planning problem by solving a sequence of subproblems.

3.8. Consider the planning problem in Figure 2.8(b). At s_0, suppose GBFS needs to choose between the actions a_1 = unstack(c,a) and a_2 = pickup(b). Let $s_1 = \gamma(s_0, a_1)$ and $s_2 = \gamma(s_0, a_2)$. Compute each of the following pairs of heuristic values, and tell whether or not they will produce the best choice:

(a) $h^{\mathrm{FF}}(s_1)$ and $h^{\mathrm{FF}}(s_2)$. (b) $h^{\mathrm{RL}}(s_1)$ and $h^{\mathrm{RL}}(s_2)$.
(c) $h^{\mathrm{add}}(s_1)$ and $h^{\mathrm{add}}(s_2)$. (d) $h^{\max}(s_1)$ and $h^{\max}(s_2)$.

3.9. Here is a state-variable version of the problem of swapping the values of two variables. The ontology of typed objects is *Objects* = *Variables* ∪ *Numbers*, *Variables* = {foo, bar, baz}; and *Numbers* = $\{0, 1, 2, 3, 4, 5\}$. There is one action schema:

$$\begin{aligned}&\mathsf{assign}(x_1, x_2, n)\\ &\quad \text{pre: } \mathsf{value}(x_2) = n\\ &\quad \text{eff: } \mathsf{value}(x_1) \leftarrow n\end{aligned}$$

where $x_1, x_2 \in$ *Variables* and $n \in$ *Numbers*. The initial state and goal are

$$\begin{aligned} s_0 &= \{\mathsf{value(foo)} = 1, \mathsf{value(bar)} = 5, \mathsf{value(baz)} = 0\};\\ g &= \{\mathsf{value(foo)} = 5, \mathsf{value(bar)} = 1\}.\end{aligned}$$

At s_0, suppose GBFS needs to choose between the actions a_1 = assign(baz,foo,1) and a_2 = assign(foo,bar,5). Let $s_1 = \gamma(s_0, a_1)$ and $s_2 = \gamma(s_0, a_2)$. Compute each of the following pairs of heuristic values, and tell whether they will produce the best choice:

(a) $h^{\mathrm{FF}}(s_1)$ and $h^{\mathrm{FF}}(s_2)$. (b) $h^{\mathrm{RL}}(s_1)$ and $h^{\mathrm{RL}}(s_2)$.
(c) $h^{\mathrm{add}}(s_1)$ and $h^{\mathrm{add}}(s_2)$. (d) $h^{\max}(s_1)$ and $h^{\max}(s_2)$.

3.10. For the planning problem in Exercise 3.9, let π be the partial plan in Figure 3.16.

(a) In π, how many threats are there? What are they? What are their resolvers?

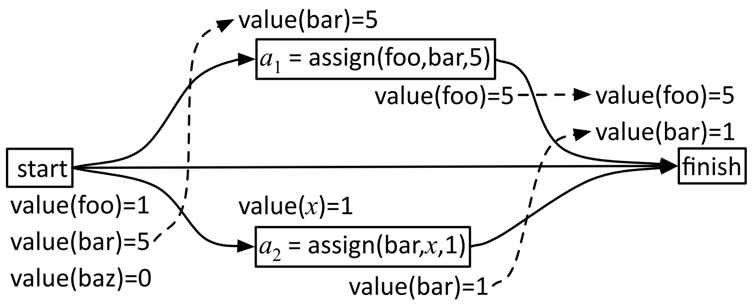


Figure 3.16. Partial plan for Exercise 3.10.

(b) Can PSP generate π? If so, describe an execution trace that will produce it. If no, explain why not.

(c) In PSP's search space, how many immediate successors does π have?

(d) How many solution plans can PSP produce from π?

(e) How many of the preceding solution plans are minimal?

(f) Trace the operation of PSP on π. Follow whichever of PSP's execution traces finds the shortest plan.

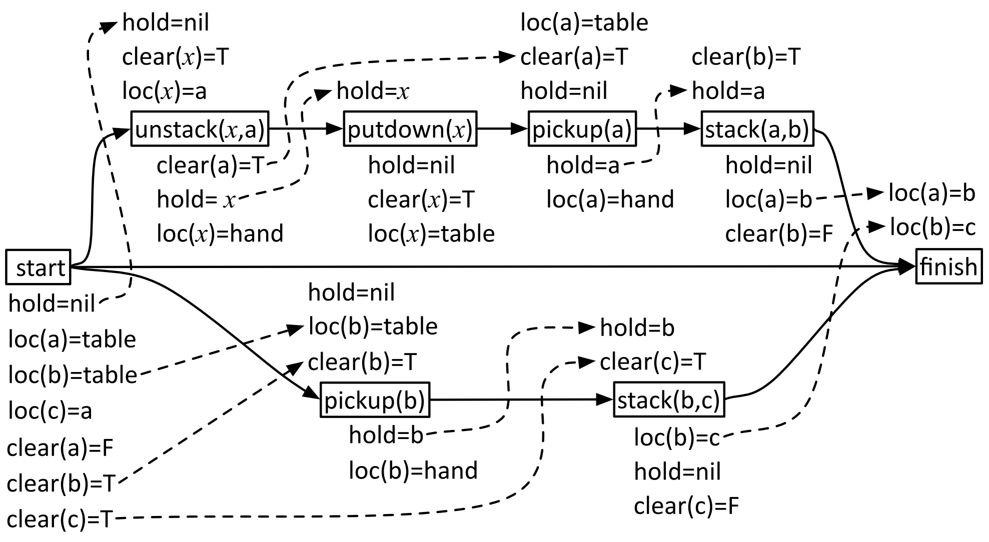


Figure 3.17. Partial plan for Exercise 3.11.

3.11. Repeat Exercise 3.10 using the planning problem in Figure 2.8(b) and the partial plan in Figure 3.17.

3.12. Let π be a partially ordered solution for a planning problem $P = (\Sigma, s_0, g)$.

(a) Write a simple modification of Run-Lazy-Lookahead to execute π.

(b) Suppose your procedure is executing π, and let π' be the part of π that it has not yet executed. Suppose an unanticipated event invalidates some of the total orderings of π' (i.e., not all of them will still achieve g). Write an algorithm to choose a total ordering of π' that still achieves g, if one exists.

3.13. If $\pi = \langle a_1, \ldots, a_n \rangle$ is a solution for a planning problem P, other orderings of the actions in π may also be solutions for P.

(a) Write an algorithm to turn π into a partially ordered solution.

(b) Are there cases in which your algorithm will find a partially ordered solution that PSP will miss? Are there cases in which PSP will find a partially ordered solution that your algorithm will miss? Explain.

4 Learning Deterministic Models

In this chapter we focus on two key topics for learning with deterministic models: learning heuristics that can speed up the search for a solution plan and the automated synthesis of the model itself.

In Section 4.1, we deal with the problem of learning heuristics that allow us to explore parts of the search space that are more likely to lead to solutions. Indeed, heuristic functions (see Section 3.2) have been demonstrated to play a key practical role in finding (optimal) plans and allowing plan generation algorithms to scale up to large state spaces. Heuristic functions can be of two different kinds: Domain independent heuristics that can be applied to any deterministic model, and domain dependent heuristics that exploit the specific structure and knowledge about the domain. Here we provide some basic techniques that, when applied to a given domain, learn domain (and problem) dependent heuristics.

In Section 4.2, we address the problem of learning a deterministic model, and we focus on learning action schemas (see Section 2.3.2). Indeed, acting and planning requires the specification of planning domains through action schemas, i.e., a lifted representation of actions with their preconditions and effects. The automated learning of action schemas is widely recognised as a key and compelling challenge to overcome the difficulties of specifying actions, which is often a time consuming and error-prone task. We discuss two main approaches to learning action schema: offline learning from a given set of traces (i.e., sequences alternating actions and states) and online learning, i.e., learning by applying actions step by step. We show how some basic routines for offline learning can be re-used for online learning. We also discuss some algorithms that use planning to learn the model online.

4.1 Learning Heuristics

We have introduced heuristic functions in Chapter 3 (see Section 3.2): a heuristic function h computes an estimate of the minimum cost $h^*(s)$ of getting from s to a goal state. If the cost is uniform or not specified, it computes an estimate of the minimum distance from a state to the goal. Heuristic functions have been demonstrated to play a key practical role in finding (optimal) plans and allowing plan generation algorithms to scale up to large state spaces. Heuristic functions can be of two different kinds:

- *Domain independent heuristics* are general, they can be applied to any deterministic model. Sections 3.2.1, 3.2.3, and 3.2.4 define three main domain independent heuristics that have been proven experimentally to be effective in several different deterministic models.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- *Domain dependent heuristics* can be more effective than domain independent heuristics, since they exploit the specific structure and knowledge about the domain. However, they are less general, they can work just in the case of a specific deterministic model, and defining them may be not obvious, sometimes requiring to elicit knowledge from domain experts.

While domain dependent heuristics may be difficult to define by hand, they can be learned automatically, e.g., through running a simulator. A good strategy can be therefore to combine both approaches, e.g., refining and improving the heuristic function by learning a domain dependent heuristic starting from a domain independent one.

An interesting idea is to devise a ”domain independent learning technique” that, when applied to a given domain, learns a domain dependent heuristics (Section 4.1.1). Such a learning technique applies in general, independently of the initial heuristic it starts from: given the constraint that the heuristic is zero in the goal states, $h(s) = 0 \; \forall s \in S_g$, we can assign an arbitrary value to h in all non-goal states. Alternatively, we can easy the task of learning domain dependent heuristics by initializing the value of h with any kind of “good” domain independent heuristic, e.g., those defined in Sections 3.2.1, 3.2.3, and 3.2.4. There is a bunch of work on learning domain independent heuristics. We discuss this work in Section 4.3.

4.1.1 Learning domain dependent heuristics

Section 4.1.1 presents LRTA* (Learning Real Time A*), which interleaves planning, learning the heuristic, and acting. LRTA* needs repeated trials to converge to optimal solutions. Next, Section 4.1.1 presents a general schema that automatically repeats the learning of h until it finds an ϵ-optimal solution. The learned heuristic h isn’t just domain-dependent, it’s problem-dependent.

Learning Real Time A*

LRTA*, Algorithm 4.1, interleaves searching, acting and learning an heuristic. During search, until the goal is reached, in each state s, LRTA* finds the action a that minimizes $Q(s,a)$, i.e., the estimated distance from s to the goal by taking into account the cost of applying a in s and the heuristic of the next state s' obtained by applying a. LRTA* updates the heuristic in s with the cost of applying a plus the heuristic in the next state s'.

If the goal is reachable from the initial state s_0, then LRTA* is guaranteed to reach the goal. Because LRTA* does a greedy best-first search (see Section 3.1.6), a single run of LRTA* does not guarantee an optimal plan. However, if h is admissible, then repeated runs of LRTA* will eventually converge to an optimal plan. A “good” initial h_0 (see e.g., Section 3.2) speeds up the convergence.[1]

[1] LRTA* generalizes well to planning with probabilistic models (see Section 9.5.3).

```
LRTA*($\Sigma, s_0, S_g, h_0$)
    Initialize the current state and the empty plan: $s \leftarrow s_0$; $\pi \leftarrow \langle\rangle$
    Initialize the heuristic: $h(s) \leftarrow h_0(s)$ for all states of $\Sigma$
    while $s \notin S_g$ do
        foreach $a \in Applicable(s)$ do
            $Q(s,a) \leftarrow \text{cost}(s,a) + h(\gamma(s,a))$
        $h(s) \leftarrow \min_a\{Q(s,a)\}$
        $a \leftarrow \text{argmin}_a\{Q(s,a)\}$
        $\pi \leftarrow \pi \cdot a$
        $s \leftarrow \gamma(s,a)$
```

Algorithm 4.1. Learning Real Time A*.

Learning ϵ-optimal heuristics

In this section, we present algorithms that learn heuristics that allow for ϵ-optimal solutions in one single run. Like LRTA*, they learn heuristics by interleaving search with updating of h. The main difference is that they keep memory of all visited states, and update h not only in the current state but in all the visited states. This provides the ability to change plan if we are not on a path to an optimal solution. Intuitively, the condition for reaching an optimal solution is that the update of h is lower than ϵ.[2]

We start by introducing some notions that allow us to keep trace of all visited states along different paths generated during the search by our algorithms. We let *Fringe* be the set of *fringe states*, which have been generated but not yet expanded; and *Interior* be the set of *interior states*, which have been expanded.[3] We let *Envelope* be the set of states that have been generated at some point by a search algorithm, that is, $Envelope = Interior \cup Fringe$.

The idea is to learn h in a state $s \in Fringe$ in a similar way to LRTA*, that is, by expanding $s \in Fringe$ and finding its successor state $\gamma(s,a)$ for all actions $a \in Applicable(s)$, and updating $h(s)$ with the value obtained from the *most promising* action at s, that is, the action in $Applicable(s)$ that minimizes the distance to the goal.

After a state $s \in Fringe$ is expanded, it becomes an interior state, i.e., $s \in Interior$. Since we aim at ϵ-optimal solutions, also the $h(s)$ of interior states must be updated with the value obtained from the most promising action at s. Each time $h(s)$ changes, each state that leads to s (that is, each s' such that $s = \gamma(s', a)$) will need its heuristic function $h(s')$ updated, because its most promising action may have changed.

Algorithm 4.2 is a general schema[4] for an algorithm that interleaves the application of actions in *Fringe* (the *expand* phase) with the updating of h in expanded and interior states (the *update* phase). The learning is performed starting from a given

[2] The main ideas presented in this section come from the application of *value iteration* (see Section 9.1.3). Here we have the specific case of value iteration for deterministic models.

[3] This is similar to Algorithm 3.2 in Section 3.1, in which *Frontier* is a set of nodes (π, s) in which s has been generated but not yet expanded, and *Expanded* is a set of nodes (π, s) in which s has already been visited. *Fringe* is the set of states in *Frontier*, and *Interior* is the set of states in *Expanded*.

[4] Algorithm 4.2 is an adaptation to deterministic domains of Algorithm 9.6 in Section 9.2.

initial heuristic function $h_0(s) = 0$ for all s that are goal states, $h_0(s) > 0$ for all the other states. The learned heuristic is problem-dependent, because it depends on the goal to be achieved.

```
Expand&Update($\Sigma, s_0, S_g, h_0$)
    initialize $h$ with $h_0$, open and fringe states with $s_0$, and $\pi$ with the empty
      plan
    until $s_0$ is solved do
1       select an open state $s$ in $\widehat{\gamma}(s_0, \pi)$
2       if $s$ is a fringe state then expand $s$
3       else
            $h(s) \leftarrow \min_{a \in Applicable(s)} [\text{cost}(s,a) + h(\gamma(s,a))]$
            $\pi \leftarrow \pi \cdot \text{argmin}_{a \in Applicable(s)} [\text{cost}(s,a) + h(\gamma(s,a))]$
```

Algorithm 4.2. Expand&Update schema.

Recall that $\widehat{\gamma}(s, \pi)$ is the *transitive closure* of a plan π on a state s, i.e., the sequence of states generated by plan π from s (see Equation 2.8). We define *solved* and *open* states as follows:

- A state $s \in Envelope$ is *open* if it is not a goal state and either it is a fringe state or it is an interior state such that $residual(s) = |h(s) - \min_a\{Q(s,a)\}| > \epsilon$. Thus a non-goal state $s \in Envelope$ is open if $(s \in Fringe) \vee (s \in Interior \wedge residual(s) > \epsilon)$.
- A state $s \in Envelope$ is *solved* if the current $\widehat{\gamma}(s, \pi)$ has no open states, that is, if $\forall s' \in \widehat{\gamma}(s, \pi)$ either $s' \in S_g$ or $residual(s') \leq \epsilon$.

The selection of an open state (line 1) must ensure that no state in $\widehat{\gamma}(s_0, \pi)$ remains open indefinitely without being chosen for revision. Moreover, the expansion of a fringe state (Line 2) generates more than one new fringe state, it is intended to apply all applicable actions to the current state s and to add them to *Fringe*. These are two main differences with respect to LRTA*, where *(i)* just one fringe state is generated at each run, thus precluding the possibility to take alternative paths to the goal, and *(ii)* open states that are not fringe states and that do not lead to the minimal cost (distance) to the goal are not updated with a new value of h and a new plan. All of this is needed to generate ϵ-optimal solutions in one run.

The expansion of a fringe state changes the current plan π and hence $\widehat{\gamma}(s_0, \pi)$. At any point, either a state s is open, or s is expanded in an open fringe state (whose update will later make s open - Line 3), or s is solved. In the latter case, $\widehat{\gamma}(s, \pi)$ does not change anymore. Algorithm 4.2 iterates until s_0 is solved, that is, there is no open state in $\widehat{\gamma}(s_0, \pi)$. With an admissible heuristic function, Algorithm 4.2 converges to a solution which is asymptotically optimal with respect to ϵ.

4.2 Learning Action Specifications

Acting and planning with (deterministic) models require the specification of actions thorough their preconditions and effects. However, the manual specification of actions is often an inaccurate, time-consuming, and error-prone task. Moreover, most often, it is impossible to specify a complete and correct model of the world. Finally, most of the times a model needs to be updated and adapted to a changing environment.

The automated learning of action specifications is widely recognised as a key and compelling challenge to overcome these difficulties. To ensure the generality and re-usability of the specification of actions, preconditions and effects are represented with lifted action schemas (see Section 2.3.2), which are independent from the specific set of objects involved in each planning domain.

Intuitively, the automated learning of action specifications is achieved by an actor that applies actions in the environment or through a simulator. The actor learns action specifications by observing the result of action applications.

The assumption that the actor does not know the model is similar to what happens in reinforcement learning in the case of probabilistic models (see Chapter 10). However, here the problem is different. The goal is not to learn a plan or a policy. The goal is to learn the deterministic model, i.e., the (lifted) specification of actions through their preconditions and effects.

In this chapter, we consider two different kinds of problems and approaches to learning actions:

- *Learning actions offline* by analyzing a given and fixed set of traces, i.e., a set of sequences alternating actions and states resulting from their application.
- *Learning actions online*, i.e., step by step by choosing an action to apply in the current state, observing the result, and iteratively choosing and applying an action in the reached state.

We will study the problem of learning action schemas in the case of full observability, i.e., when an actor has access to the value of all state variables in each state. See Section 4.3 for a discussion on the problem of learning actions from observations in which the value of some state variable is unknown, and/or some of the executed actions might be missing.

The structure of the remaining chapter is as follows. In Section 4.2.1, we address the problem of learning actions offline: we start from the hypotheses that all actions in the trace are applicable. We then drop such assumption, allowing for inapplicable actions in the trace, a key step for extending the learning algorithm in the case of learning action models online. Indeed, in the online case, the actor chooses actions to be applied step by step and it cannot know whether actions are applicable.

In Section 4.2.2, we address the problem of learning actions online, i.e., the case in which an actor applies actions in the current observed state and observes the results of such applications. Before addressing the online learning action problem, we consider first the case of "learning by queries". In this scenario, the actor chooses a state and an action and gets the result of applying such action in such state. The intuition is to query a platform that acts as an oracle and replies to the query with the results of action applications. We introduce here an important concept that is essential for addressing the online learning problem: the notion of *informative state-action pair*. Intuitively, a state and an action are informative when the application of that action in that state allows the actor to learn something new about preconditions and effects. This notion is important since a learning algorithm can guide the application of actions to informative states. Moreover, the algorithm can know if there is anything more to learn, or if it should terminate its task.

Once we have provided such basic concepts and routines, we address the problem of learning actions online. Differently from learning by queries, in this case the actor can not choose states arbitrarily at each step, but it must chose actions from the current state. This is a much more realistic hypotheses than the assumption in learning by query, since it mimics what happens in learning by acting in the real world, and it deals with the problem of learning dynamically by the application of actions in an unknown environment. Also in this case, the actor exploits the key notion of informative state-action pair by looking for states where it can apply actions that allow for learning something new. We devise an approach where the actor plans for reaching an informative state where to apply an action in the current model. The application of actions in the generated plan may fail, since the actor generates plans with an incomplete and imperfect model, however the actor can learn also from failures.

4.2.1 Offline Action Learning

We recall the definition of (lifted) state variable, (lifted) assignment, and action schemas (or action template) given in Section 2.3. *(Ground) state variables* are expressions of the form $x(c_1, \dots, c_n)$, also written $x(\mathbf{c})$, where $c_1, \dots, c_n$ are constants denoting typed objects. *Lifted state Variables* are expressions of the form $x(z_1, \dots, z_n)$, also written $x(z)$, where x is a *state variable name* and $z_1, \dots, z_n$ are parameters of given types. *(Ground) assignments* are expressions of the form $x(c_1, \dots, c_n) = c_{n+1}$, where x is a state-variable name and $c_1, \dots, c_n, c_{n+1}$ are constants denoting typed objects. *Lifted assignments* are expressions of the form $x(z_1, \dots, z_n) = z_{n+1}$, where x is a state-variable name and $z_1, \dots, z_n, z_{n+1}$ are parameters.

We recall the definition of action schema (see Definition 2.7 in Section 2.3.2): $a(z_1, \dots, z_k)$, also written $a(\boldsymbol{z})$, is the head of an action schema with a the action name and $z_1, \dots, z_k$ a list parameters. We write $a(c_1, \dots, c_k)$ (also written $a(\boldsymbol{c})$) as the head of an action grounded with the constants $c_1, \dots, c_k$ in place of the parameters $z_1, \dots, z_k$. We call $a(c_1, \dots, c_k)$ (also written $a(\boldsymbol{c})$) a ground action name. Preconditions and effects are sets of lifted assignments.

We extend the notion of lifted state variable and lifted assignment by allowing for a set of *special constants* which contains special symbols that denote constant values that are generally used in any planning domain. For instance, we suppose we have two special constants true and false of type *Bool* that denote the Boolean values "true" and "false". Indeed, in the case of Boolean state variables, we prefer to use the notation $x(\boldsymbol{c}) = \mathsf{true}$ and $x(\boldsymbol{c}) = \mathsf{false}$ rather than $x(\boldsymbol{c})$ and $\neg x(\boldsymbol{c})$, since this will allow us to use a uniform representation for Boolean and non-Boolean state variables in the learning algorithms.

The input to the problem of learning actions offline is a set of finite *traces*. A trace is a sequence $s_0, a_1, s_1, a_2, \ldots, a_n, s_{n+1}$ of alternating states and action names with their grounded parameters. The set of traces can be broken into a set T of *transitions* of the form $(s, a(\boldsymbol{c}), s')$.[5] Given a set of transitions T (or equivalently a set of traces), we want to learn the preconditions and effects (sets of lifted assignments) of the actions that appear in T.[6]

We will start our approach to learning action schemas based on some simple rules that are valid for ground assignments and actions.[7] Let $\mathrm{pre}(a(\boldsymbol{c}))$ and $\mathrm{eff}(a(\boldsymbol{c}))$ be the preconditions and effects of $\mathrm{pre}(a(\boldsymbol{z}))$ and $\mathrm{eff}(a(\boldsymbol{z}))$, resp., grounded with constants $\boldsymbol{c}$ (i.e., obtained by replacing parameters $\boldsymbol{z}$ with constants $\boldsymbol{c}$ of the proper type). Let $(s, a(\boldsymbol{c}), s') \in T$ be a transition, where s and s' are states, and $a(\boldsymbol{c})$ is a ground action. Given a ground transition $(s, a(\boldsymbol{c}), s')$, the following rules state when a ground assignment $x(\boldsymbol{c}) = c$ can be a ground precondition or a ground effect of a ground action name:

1. If $x(\boldsymbol{c}) = c \notin s$, then $x(\boldsymbol{c}) = c \notin \mathrm{pre}(a(\boldsymbol{c}))$
2. If $x(\boldsymbol{c}) = c \notin s'$, then $x(\boldsymbol{c}) = c \notin \mathrm{eff}(a(\boldsymbol{c}))$
3. If $x(\boldsymbol{c}) = c \in s' \setminus s$, then $x(\boldsymbol{c}) = c \in \mathrm{eff}(a(\boldsymbol{c}))$

From these simple rules we can define the upper and lower bounds of grounded preconditions and effects, given a set of transitions (traces) T.

$$\emptyset \subseteq \mathrm{pre}(a(\boldsymbol{c})) \subseteq \bigcap_{(s,a(\boldsymbol{c}),s')\in T} s \tag{4.1}$$

$$\bigcup_{(s,a(\boldsymbol{c}),s')\in T} s' \setminus s \subseteq \mathrm{eff}(a(\boldsymbol{c})) \subseteq \bigcap_{(s,a(\boldsymbol{c}),s')\in T} s' \tag{4.2}$$

Equation 4.1 holds since a ground assignment cannot be a precondition if it is not in every state s where the action $a(\boldsymbol{c})$ is applied, and therefore only assignments that are in all the states where action $a(\boldsymbol{c})$ is applied may be preconditions of $a(\boldsymbol{c})$. On the other hand, the fact that a state variable has been observed in a state of the trace, does not necessarily means that it is a precondition. It may be even the case that action $a(\boldsymbol{c})$ has no precondition at all, and therefore the lower bound of $\mathrm{pre}(a(\boldsymbol{c}))$ is the empty set.

Equation 4.2 states that an assignment cannot be an effect if it is not in all the states resulting from the application of an action. On the other hand, all the state variables that have an assignment that in the resulting state s' is different from the one in the original state s are necessarily effects, and this is an upper bound for $\mathrm{eff}(a(\boldsymbol{c}))$.

[5]Indeed, the order of transitions in the trace does not influence the learning task. This is true in the cased of full observability, in which there are not *hidden state variables*, i.e., the actor has access to the value of all state variables in all states. In Section 4.3, we will briefly discuss works that deal with *partial traces* with hidden state variables, where the order of transitions in a trace is important.

[6]With abuse of notation, we use T both for the trace and the set of transitions given in input to the learning problem.

[7]In the literature, these rules are called the *Safe Action Model (SAM) rules*.

Given these simple considerations, we can devise a first simple algorithm for computing the lifted preconditions and effects from a set of transitions. Let T be a set of transitions $(s, a(\boldsymbol{c}), s')$. Let $s(\boldsymbol{c})$ denote the set of ground assignments in s that contain only constants in $\boldsymbol{c}$. $s(z)$ is the result of the replacement of each c_i in $\boldsymbol{c}$ with z_i. Algorithm 4.3 simply exploits the ideas underlying rules in Equation 4.1 and Equation 4.2.

```
Action-Offline-Learning-Simple($T$)
    for $a$ action name that appears in $T$ do
        $\mathrm{pre}(a(z)) \leftarrow \bigcap_{(s,a(\boldsymbol{c}),s')\in T} \; s(z)$
        $\mathrm{eff}(a(z)) \leftarrow \bigcup_{(s,a(\boldsymbol{c}),s')\in T} \; s'(z) \setminus s(z)$
```

Algorithm 4.3. Simple Action Offline Learning.

Example 4.1. In this example, we consider the following trace: a robot is loaded, then it moves to a target location where it gets unloaded, and finally the robot moves back to the original location. The corresponding trace is the following:

$$
\begin{aligned}
T &= \langle s_0, \mathsf{load}(\mathsf{r1}), s_1, \mathsf{move}(\mathsf{r1}, \mathsf{l1}, \mathsf{l2}), s_2, \mathsf{unload}(\mathsf{r1}), s_3, \mathsf{move}(\mathsf{r1}, \mathsf{l2}, \mathsf{l1}), s_4\rangle, \text{where}\\
s_0 &= \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{false}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l1}\},\\
s_1 &= \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{true}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l1}\},\\
s_2 &= \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{true}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l2}\},\\
s_3 &= \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{false}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l2}\}\\
s_4 &= \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{false}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l1}\}
\end{aligned}
$$

Algorithm 4.3 computes the following action schema:

$\mathsf{load}(r)$	$\mathsf{unload}(r)$	$\mathsf{move}(r, l, l')$
pre : $\mathsf{loaded}(r) = \mathsf{false}$	pre : $\mathsf{loaded}(r) = \mathsf{true}$	pre : $\mathsf{pos}(r) = l$
eff : $\mathsf{loaded}(r) = \mathsf{true}$	eff : $\mathsf{loaded}(r) = \mathsf{false}$	eff : $\mathsf{pos}(r) = l'$

This example gives an intuition of the fact that Algorithm 4.3 learns preconditions that may be preconditions but are not guaranteed to be preconditions. Indeed, if the trace would have stopped at state s_3, we would have had $\mathsf{loaded}(\mathsf{r}) = \mathsf{true}$ in the preconditions of $\mathsf{move}(r, l, l')$, i.e., Algorithm 4.3 would have learned that a robot can be moved from one location to another one only when it is loaded. □

Notice that, as clearly stated by Equation 4.1, the preconditions $\mathrm{pre}(a(z))$ computed by Algorithm 4.3 are not necessarily preconditions. In the following, we will indicate with the notation $\mathrm{pre}_?(a(z))$ the fact the lifted assignments in $\mathrm{pre}_?(a(z))$ might be preconditions, but are not guaranteed to be preconditions. We say they are *potential preconditions*. We call the preconditions that are guaranteed to be preconditions, *certain preconditions*, and we write $\mathrm{pre}_!(a(z))$. On the contrary, Algorithm 4.3 computes in $\mathrm{eff}(a(z))$ all the *certain effects* (we write them as $\mathrm{eff}_!(a(z))$) that are

```
Action-Offline-Learning?!($T$)
    for $a$ action name that appears in $T$ do
        $\text{pre}_?(a(z)) \leftarrow \bigcap_{(s,a(\mathbf{c}),s') \in T} \; s(z)$
        $\text{eff}_!(a(z)) \leftarrow \bigcup_{(s,a(\mathbf{c}),s') \in T} \; s'(z) \setminus s(z)$
        $\text{eff}_?(a(z)) \leftarrow \bigcap_{(s,a(\mathbf{c}),s') \in T} \; s'(z)$
```

Algorithm 4.4. Action Offline Learning with potential and certain preconditions and effects.

guaranteed to be effects, but not the *potential effects*, that we write as $\text{eff}_?(a(z)))$. In Algorithm 4.4, we refine the simple Algorithm 4.3 for offline learning taking into account the difference between potential and certain preconditions and effects.

Example 4.2. Consider example Example 4.1. Algorithm 4.4 gives the following results:

$$\begin{aligned}
\text{pre}_?(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}\\
\text{eff}_!(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}\\
\text{eff}_?(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}
\end{aligned}$$

$$\begin{aligned}
\text{pre}_?(\mathsf{unload}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}\\
\text{eff}_!(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}\\
\text{eff}_!(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}
\end{aligned}$$

$$\begin{aligned}
\text{pre}_?(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l\}\\
\text{eff}_!(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l'\}\\
\text{eff}_?(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l'\}
\end{aligned}$$

Notice that we could eliminate the potential effects that are also certain. □

We then address the interesting case in which we have a trace that includes non-applicable actions, i.e., actions that may fail. Indeed we should have the possibility to learn also from failure, and this will be important for on-line learning, since in the online case we cannot be guaranteed that the selected actions are applicable in the current state.

Let T be a set of triples $(s, a(\mathbf{c}), s')$, where s is a state, $a(\mathbf{c})$ is an action name grounded with constants $c_1, \ldots, c_n$, and s' is either a state or failure.

The result of Algorithm 4.5 is a quadruple of $(\text{pre}_?, \text{pre}_!, \text{eff}_?, \text{eff}_!)$. Notice that Algorithm 4.5 introduces disjunctions in the preconditions, which are not allowed in the classical formulation of action schemas. We have that :

1. For any $x_1(z) = z_1 \vee \cdots \vee x_n(z) = z_n \in \text{pre}_!(a(z))$, there is an i such that $x_i(z) = z_i \in \text{pre}(a(z))$;
2. $\text{pre}(a(z)) \subseteq \text{pre}_?(a(z))$
3. $\text{eff}_!(a(z)) \subseteq \text{eff}(a(z)) \subseteq \text{eff}_?(a(z))$

Algorithm 4.5 has an important advantage with respect previous algorithms! By dealing with failures it can compute certain preconditions, i.e., preconditions that are actually needed to apply the action. This is impossible without dealing with failure.

```
Action-Offline-Learning-with-Failure($T$)
1  for $a$ action name that appears in $T$ do
2      $\mathrm{pre}_?(a(z)) \leftarrow \bigcap_{\substack{(s,a(\boldsymbol{c}),s')\in T\\ s'\neq \text{failure}}} s(z)$
3      $\mathrm{pre}_!(a(z)) \leftarrow \bigcup_{(s,a(\boldsymbol{c}),\text{failure})\in T} \bigvee(\mathrm{pre}_?(a(z)) \setminus s(z))$
4      for $\alpha \neq \beta \in \mathrm{pre}_!(a(z))$ do
           if $\alpha \models \beta$ then
5              remove $\beta$ from $\mathrm{pre}_!(a)$
6      $\mathrm{eff}_!(a(z)) \leftarrow \bigcup_{\substack{(s,a(\boldsymbol{c}),s')\in T\\ s'\neq \text{failure}}} s'(z) \setminus s(z)$
7      $\mathrm{eff}_?(a(z)) \leftarrow \bigcap_{\substack{(s,a(\boldsymbol{c}),s')\in T\\ s'\neq \text{failure}}} s'(z)$
```

Algorithm 4.5. Offline Action Learning with actions that may fail.

Example 4.3. This example is a simple extension of Example 4.1, where we start by unloading an unloaded robot, an action that is not applicable. In this example we suppose that the failing action leaves the situation unchanged, i.e., we stay in the original state. We add to the transitions that can be extracted from the trace in Example 4.1, the following transition:

$$\langle s_0 = \{\mathsf{loaded}(\mathsf{r1}) = \mathsf{false}, \mathsf{pos}(\mathsf{r1}) = \mathsf{l1}\}, \mathsf{unload}(\mathsf{r1}), \mathsf{failure}\rangle$$

Algorithm 4.5 gives the following results:

$$\begin{aligned}
\mathrm{pre}_?(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}\\
\mathrm{pre}_!(\mathsf{load}(r)) &= \varnothing\\
\mathrm{eff}_!(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}\\
\mathrm{eff}_?(\mathsf{load}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{pre}_?(\mathsf{unload}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}\\
\mathrm{pre}_!(\mathsf{unload}(r)) &= \{\mathsf{loaded}(r) = \mathsf{true}\}\\
\mathrm{eff}_!(\mathsf{unload}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}\\
\mathrm{eff}_?(\mathsf{unload}(r)) &= \{\mathsf{loaded}(r) = \mathsf{false}\}
\end{aligned}$$

$$\begin{aligned}
\mathrm{pre}_?(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l\}\\
\mathrm{pre}_!(\mathsf{move}(r,l,l')) &= \varnothing\\
\mathrm{eff}_!(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l'\}\\
\mathrm{eff}_?(\mathsf{move}(r,l,l')) &= \{\mathsf{pos}(r) = l'\}
\end{aligned}$$

Notice how dealing with action application failures allows us to determine certain preconditions. □

The learning algorithms presented so far in this section are based on the idea to compute the preconditions and effects of an action $a(\boldsymbol{c})$ by analysing all the transitions of the same action $a(\boldsymbol{c})$ in T. A different approach is to select a transition (in a trace) and to incrementally update the preconditions and effects. This alternative approach is particularly interesting, since it will allow us to provide the basic routines for online action learning. Indeed in online learning (Section 4.2.2), we must incrementally select an action an analyse the transition resulting from the action application.

The incremental offline algorithms Algorithm 4.6, Algorithm 4.7, and Algorithm 4.8 are the incremental version of the offline algorithms Algorithm 4.3, Algorithm 4.4, and Algorithm 4.5.

```
Action-Incremental-Learning-Simple($T$)
    for $a$ action name that appears in $T$ do
        pre$(a(z)) \leftarrow$ U
        eff$(a(z)) \leftarrow \varnothing$
    while $T \neq \varnothing$ do
        choose $(s, a(\boldsymbol{c}), s') \in T$
        pre$(a(z)) \leftarrow pre(a(z)) \cap s(z)$
        eff$(a(z)) \leftarrow$ eff$(a(z)) \cup s'(z) \setminus s(z)$
        $T \leftarrow T \cap (s, a(\boldsymbol{c}), s')$
```

Algorithm 4.6. A simple algorithm for Incremental Action Learning.

```
Action-Incremental-Learning?!($T$)
    for $a$ action name that appears in $T$ do
        pre$_?(a(z)) \leftarrow$ U
        eff$_?(a(z)) \leftarrow$ eff$_!(a(z)) \leftarrow \varnothing$
    while $T \neq \varnothing$ do
        choose $(s, a(\boldsymbol{c}), s') \in T$
        pre$_?(a(z)) \leftarrow$ pre$_?(a(z)) \cap s(z)$
        eff$_!(a(z)) \leftarrow$ eff$_!(a(z)) \cup s'(z) \setminus s(z)$
        eff$_?(a(z)) \leftarrow$ eff$_?(a(z)) \cup s'(z)$
        $T \leftarrow T \cap \{(s, a(\boldsymbol{c}), s')\}$
```

Algorithm 4.7. Incremental Action Learning with potential and certain preconditions and effects.

```
Action-Incremental-Learning-with-Failure($T$)
    for $a$ action name that appears in $T$ do
        $\text{pre}_?(a(z)) \leftarrow \text{eff}_?(a(z)) \leftarrow \text{U}$
        $\text{pre}_!(a(z)) \leftarrow \text{eff}_!(a(z)) \leftarrow \varnothing$
    while $T \neq \varnothing$ do
        choose $(s, a(\boldsymbol{c}), s') \in T$
        if $s' \neq$ failure then
            $\text{pre}_?(a(z)) \leftarrow pre(a(z)) \cap s(z)$
            $\text{eff}_!(a(z)) \leftarrow \text{eff}_!(a(z)) \cup s'(z) \setminus s(z)$
            $\text{eff}_?(a(z)) \leftarrow \text{eff}_?(a(z)) \cap s'(z)$
        else if there is no $\beta \in \text{pre}_!(a(z))$ s.t. $\beta \models \bigvee(\text{pre}_?(a(z) \setminus s(z))$ then
            $\text{pre}_!(a(z)) \leftarrow \text{pre}_!(a(z)) \cup \{\bigvee(\text{pre}_?(a(z)) \setminus s(z)))\}$
        $T \leftarrow T \cap \{(s, a(\boldsymbol{c}), s')\}$
```

Algorithm 4.8. Incremental Action Learning with actions that may fail

4.2.2 Online Action Learning

While in offline learning we assume a set of transitions is given in input to the learning algorithms, online action learning algorithms do not have a set of transitions in input, but they must build the set of transitions by selecting actions to be applied incrementally.

We start by defining a basic building block for online learning, i.e., learning from the application of a ground action name. Algorithm 4.9 takes in input a state s, a ground action name $a(\boldsymbol{c})$, and a set of previously computed preconditions $\text{pre}(a(z))$ and effects $\text{eff}(a(z))$ of action $a(z)$. It applies the action to the state s and stores the result in s', which might be a state if the action succeeds or failure if it fails. Notice that the part of Algorithm 4.9 computing the preconditions and effects of the action application is the same as in the incremental Algorithm 4.8.

```
Learn-by-action-application($s, a(\boldsymbol{c}), \text{pre}(a(z)), \text{eff}(a(z))$)
    $s' \leftarrow$ apply action $a(\boldsymbol{c})$ to state $s$
    if $s' \neq$ failure then
        $\text{pre}_?(a(z)) \leftarrow \text{pre}(a(z)) \cap s(z)$
        $\text{eff}_!(a(z)) \leftarrow \text{eff}_!(a(z)) \cup s'(z) \setminus s(z)$
        $\text{eff}_?(a(z)) \leftarrow \text{eff}_?(a(z)) \cap s'(z)$
    else if there is no $\beta \in \text{pre}_!(a(z))$ s.t. $\beta \models \bigvee(\text{pre}_?(a(z) \setminus s(z))$ then
        $\text{pre}_!(a(z)) \leftarrow \text{pre}_!(a(z)) \cup \{\bigvee(\text{pre}_?(a(z)) \setminus s(z)))\}$
```

Algorithm 4.9. Learning by action application.

We can now define a first version of an online algorithm, where learning is performed by applying an action in a given state. We suppose we can freely select the

```
Naive-Learning-Actions-by-Queries(state variable and action names, $C$)
    for each action name $a$ do
        $\text{pre}_?(a(z)) \leftarrow \text{eff}_?(a(z)) \leftarrow$ U (the universal set)
        $\text{pre}_!(a(z)) \leftarrow \text{eff}_!(a(z)) \leftarrow \varnothing$
    for all pairs $\langle s, a(\boldsymbol{c})\rangle$ do
        Learn-by-action-application($s, a(\boldsymbol{c}), \text{pre}(a(z)), \text{eff}(a(z))$)
```

Algorithm 4.10. A naive version of Action Learning by Queries.

```
Learning-Actions-by-Queries(state variable and action names, $C$)
    for each action name $a$ do
        $\text{pre}_?(a(z)) \leftarrow \text{eff}_?(a(z)) \leftarrow$ U (the universal set)
        $\text{pre}_!(a(z)) \leftarrow \text{eff}_!(a(z)) \leftarrow \varnothing$
    while there exists an informative state-action pair $\langle s, a(\boldsymbol{c})\rangle$ do
        Learn-by-action-application($s, a(\boldsymbol{c}), \text{pre}(a(z)), \text{eff}(a(z))$)
```

Algorithm 4.11. Action Learning by Queries.

state where to apply the action (without knowing whether it will succeed or fail). It is like we can query an oracle, or a system simulating the real environment, which answers to the query with the result of the application of the selected action to the selected state. We call this approach ”learning by query”.

Algorithm 4.10 takes as input a set of state variable names, a set of action names, and a set of constants C. It represents a naive version of online learning by query in the sense that blindly applies all possible ground actions to all possible states. There is no attempt to select a pair action-state that could provide useful information to learn the preconditions and effects. It does not exploit the real advantage of the online approach, i.e., the fact that we can choose the action to apply in a given state, and therefore we can choose an *informative state-action pair*, i.e. a state and a ground action that allow the actor to learn preconditions and effects that have not been learned yet. The intuition is that, during the learning process, a state-action pair is informative when the application of the action to the state provides some further useful information, thus allowing us to learn something more with respect to what we have learned so far.

During the learning process, a *state-action pair* $\langle s, a(\boldsymbol{c})\rangle$ *is informative* if all the certain ground preconditions $\text{pre}_!(a(\boldsymbol{c}))$ hold in s, and at least one among the potential preconditions $\text{pre}_?(a(\boldsymbol{c}))$ or one among the potential effects $\text{eff}_?(a(\boldsymbol{c}))$ does not hold in s.

Intuitively, a state is informative for a given action if all the certain preconditions of the action hold in the state so that we have a chance to apply the action; at least one potential precondition/effect does not hold, so that we can understand whether it is a certain precondition/effect or it is not a precondition/effect at all.

Given the notion of informative state-action pair, we can define Algorithm 4.11 that learns by querying the results of applying actions to states that allow learning something more, until no further informative state-action pair exists.

We are now going to drop a basic assumption underlying the approach in Algorithm 4.11 to online learning by querying the results of applying a given action in a given state. In the new scenario, we cannot select any state and query for the results of applying an action in that state. We can instead sense the current state, and we can then select an action to be applied in the current state. If the action succeeds, we get to the state resulting from the application of the action. If the action fails, we get to a state possibly different from the foreseen state. It maybe the same original state in which we have applied the action, or another state, but we suppose we can go on with the learning process.

```
Online-Action-Learning(state variable and action names, $C$)
    for each action name $a$ do
        $\mathrm{pre}_?(a(z)) \leftarrow \mathrm{eff}_?(a(z)) \leftarrow$ U (the universal set)
        $\mathrm{pre}_!(a(z)) \leftarrow \mathrm{eff}_!(a(z)) \leftarrow \varnothing$
    plan $\pi \leftarrow \langle\rangle$
    $s \leftarrow$ observe the current state
    while True do
        if $\pi = \langle\rangle$ then
1           $\pi \leftarrow$ a plan that leads from $s$ to an informative $\langle s', a(\boldsymbol{c})\rangle$
            if $\pi = nil$ then
                return (no informative pair state action is reachable from $s$)
            $\pi \leftarrow \pi \cdot a(\boldsymbol{c})$
        $a(\boldsymbol{c}) \leftarrow$ pop($\pi$)
        Learn-by-action-application($s, a(\boldsymbol{c}), \mathrm{pre}(a(z)), \mathrm{eff}(a(z))$)
        if the application fails then $\pi \leftarrow \langle\rangle$
        $s \leftarrow$ observe the current state
```

Algorithm 4.12. Online Learning of Action Models.

Algorithm 4.12 takes in input a set of state variable names, a set of action names, and a set of constants C. It exploits the notion of informative state-action pair. However, it cannot freely select an informative state-action pair, like in the query based approach (Algorithm 4.11). It has to try to reach a state s where it can apply the action $a(\boldsymbol{c})$ such that $\langle s, a(\boldsymbol{c})\rangle$ is an informative state-action pair. Planning at Line 1 is performed in a model with all the potential and certain preconditions that have been computed at the moment. If no plan for an informative state-action pair exists, then from the current state there is no reachable state such that we have an informative state-action pair. This means there is nothing more we can learn, and the Algorithm 4.12 returns. If a plan exists, then we add to the current plan the action $a(\boldsymbol{c})$ of the informative state-action pair, and we call the subroutine learning by applying the first action in the plan to the current state (Algorithm 4.9). If the application fails, we plan again for a different state where we have an informative state-action pair. If the action succeeds, we go on by applying the next action in the plan. Notice that, if we are so lucky that all actions in the plan are applicable, then the plan becomes empty, and we start by planning again to reach a next state where we have an informative state-action pair.

Notice a further difference of Algorithm 4.12 with learning by query (Algorithm 4.10 and Algorithm 4.11): the former constructs automatically a single trace of action applications that may fail, while learning by query constructs a set of transitions that are not necessarily connected in a trace.

Algorithm 4.12 is an example of "planning to learn", in the sense that it plans to try to reach an informative state-action pair. In spite of the fact that it plans in an incomplete or even incorrect model, and the applications of actions of the generated plans can fail, it uses planning to try to reach a state where we can apply an action and learn new preconditions and effects. The main difference w.r.t. the "planning to learn" paradigm depicted in Figure 1.2 is that learning and planning are tightly integrated, since while we learn, we plan to learn from each single action application, rather than collecting a set of training examples.

4.2.3 Comparing Offline and Online

Offline learning is the most common approach in research literature. The underlying idea is that we have a log of plan executions, the corresponding traces or the correspondng set of transitions, and from that log we learn what we can. Notice that the offline learning algorithms presented in Section 4.2.1 guarantee to learn preconditions and effects in the lower and upper bound defined by rules Equation 4.1 and Equation 4.2. They are correct and complete with respect to the input trace, in the intuitive sense that they do not learn an effect that is not an effect of an action and they do not eliminate a precondition that is a precondition of the action. However, what they can learn is limited to the specific set of transitions given in input. If the set of transitions does not provide all the useful information, there is nothing offline learning can do.

Online action learning is a different approach where we learn incrementally step by step by applying actions in some states. Intuitively, the two algorithms that learn by querying (Algorithm 4.10 and Algorithm 4.11) are correct and complete in a stronger sense than the correctness and completeness guaranteed by offline algorithms. Indeed they can query all possible state-action pairs and learn only and all the preconditions and the effects that are actually preconditions and effects of an action. This in theory. In practice, most often, the space of state-action pairs is huge, and cannot be explored exhaustively.

The online algorithm that drops the assumption of freely selecting a state to apply an action (Algorithm 4.12), and has instead to sense the current state and select an action to apply in that state, does not guarantee to reach all the possible states of the state space, like "learning by querying" does. Indeed, it depends on which is the initial state that is sensed, since there might be states that are unreachable from such state. Moreover, some actions might not be reversed (simple dead end), or it may end up in a complex dead end (i.e., a state that leads to a loop that does not allow the algorithm to get out of that loop). Algorithm 4.12 guarantees to learn all and only the preconditions and effects with respect the reachable states in the case of safely explorable models (there are no irreversible actions or loops without a possibility to exit from the loop), i.e., it learns everything it can learn in a model with only the reachable states from the initial sensed state.

In spite of the fact that learning by query algorithms have the possibility to explore the whole space of state-action pairs, Algorithm 4.12 that senses the current state and that can apply actions only in the current state is much more interesting. On the one hand, it pursues a more difficult approach, due to the fact that it *"cannot jump arbitrarily from one state to another"*; it can only apply actions in the current state where the actor is. As a consequence, it can *try* to reach an informative state-action pair by planning, but there is no guarantee to reach it since the model that has been learned so far can be incomplete and incorrect.

On the other hand, it works in a scenario that is much more realistic, where it is possible to learn from applications of actions in a real environment, and can be used to incrementally and dynamically learn in a (partially) unknown environment. The choice of the action to apply is an important step. We can indeed interleave a learning phase with an exploration phase that selects the actions to apply. We could use different heuristics for the exploration phase in the style of what is done in reinforcement learning for probabilistic domains, see Chapter 10. However, such techniques do not exploit the idea of informative states, and therefore they are not devoted to generate *informative traces*, i.e. traces leading to informative states. It is true that, since we do not know whether actions are applicable, in the case of online learning by planning (Algorithm 4.12), we are not guaranteed to reach an informative state, but the important point is that we can try to get there and learn also from failure. In this way, we can guarantee that we get to all informative states that are reachable.

4.3 Discussion and Bibliographic Notes

Since the seminal work on learning for planning in deterministic domains [360, 203, 1124, 421, 1124, 1126, 580, 331, 574], and the first approaches to learning action models by integrating learning, planning, and execution [382], research in learning for planning has addressed different kinds of problems. For instance, the work in [247] focuses on learning action-sequences as macro-actions to use them as an heuristic during search; [579] introduces the notion of explanation based learning, which involves using prior knowledge to explain why training examples have given some labels, and uses this explanation to guide the learning. In [1206], a model-lite approach is proposed to do planning, where a planner is supposed to work with an incomplete model and using a probabilistic approach to learn and update the model. See [1236, 556] for general reviews of machine learning for planning.

In the following sections we focus on recent work on learning heuristics (Section 4.3.1) and learning action specifications (Section 4.3.2).

4.3.1 Learning Heuristics

In Section 4.1, have shown a way to learn domain dependent heuristics using techniques based on value iteration (see Section 9.1.3 in Chapter 9) and adapting them to deterministic models. Algorithm 4.1 (LRTA*) has been devised in [639]. There has been indeed a lot of work on learning heuristics or value functions to control the search (see, e.g., [174, 1225, 191]).

Recent works have addressed the task of improving domain independent heuristics by exploiting the notion of relaxed-plan for STRIPS domains. The works in [1207, 1210] use machine learning to improve given domain independent heuristics. They use linear regression to learn the difference between the actual distance-to-go and the estimate given by a relaxed-plan heuristic. Each feature in the feature space for linear regression is an integer valued function of the state, the goal, and the set of actions. The feature space strongly correlates with the length of the shortest plan to the goal, and improves the heuristic w.r.t. the one purely based on relaxed plans, which ignores delete-lists and often underestimates the distance to the goal. The work in [1196] builds on [1207] by incorporating ideas from structural prediction and exploiting the power of discriminative machine learning approaches. The learning method discriminates between "good" and "bad" states rather than attempting to precisely model the distance to the goal. It takes into account the actual search performance of the heuristic during the learning phase by iteratively updating the heuristic in response to observed search errors.

Greedy heuristic search performance in several combinatorial search domains are investigated in [1179]. Their results suggest that heuristics that exhibit strong correlation with the distance-to-go are less likely to produce large local minima. The work in [1180] makes use of the Kendall rank correlation coefficient to select a pattern database. The work in [385] improves heuristics in a greedy best-first search exploiting the ordering of states induced by the original heuristics rather than ordinary least-squares regression.

While most of the works based on relaxed plans improve existing heuristics, the approach presented in [1006] learns domain independent heuristics from scratch. It is based on a deep learning recurrent encode-decode neural network based on hypergraph networks, i.e., a generalization of graph networks to hyper-graphs, induced by the delete-relaxation relaxed plans. The work in [222] extends the results in [1006] with grounded and lifted graph representations of planning tasks suitable for learning domain-independent heuristics, learning in this way heuristics that allow planners to solve large problems. Other recent works make use deep learning techniques. For instance, [972] combines multiple heuristics values by using a neural network whose features are the different available heuristics. In a different approach, [49] generates a sequence of heuristics from a given weak heuristic and a set of unsolved training instances. The training instances that can be solved using the weak heuristic provide training examples for a learning algorithm that produces a heuristic that is expected to be stronger than the weak heuristic. If the weak heuristic cannot solve any of the given instances, random walks create a sequence of successively more difficult training instances starting with ones that are guaranteed to be solvable by the weak heuristic. The process is then repeated, producing a sequence of heuristics until a sufficiently strong heuristic is produced.

The work in [1093] proposes an interesting and effective approach that learns heuristics online, during search, without requiring expensive pre-training. The works in [346, 730] use imitation learning to generalize neural networks heuristics over the states in the state space of the given instance. In [347], heuristics are learned from scratch using states as the neural network input, Generalized heuristics can be learned in the absence of symbolic action models using deep neural networks that utilize an input predicate vocabulary but are agnostic to object names and quantities [585]. Potential heuristics are introduced in [912]. They represent heuristics as a linear combination of state features, whose evaluation is performed by summing the weights of all features that are true in a state. The computational complexity of potential heuristic synthesis for satisficing planning in studied in [497]. The work shows that the problem of synthesizing a dead-end avoiding potential heuristic is PSPACE-complete and thus as hard as planning.

The learning of heuristics for classical planning is an active field of research, and a workshop on Heuristics and Search for Domain-independent Planning (HSDIP) is dedicated every year at the main conference on planning (ICAPS).

4.3.2 Learning Action Specifications

Several works have addressed the task of learning action specifications and have provided important results from different perspectives and according to different assumptions. We structure this section by discussing existing work in learning action preconditions and effects according to the offline and the online approach (Section 4.3.2 and Section 4.3.2, respectively). We conclude the section by discussing some first attempts to learn deterministic action specifications from continuous perceptions (Section 4.3.2).

Offline approaches

A lot of research has been done on learning action specifications offline. The rules in Equation 4.1 and Equation 4.2, which specify the lower and upper bounds for ground preconditions and effects, and which provide the basis for the offline algorithms presented in Section 4.2.1, are called SAM (*Safe Action Model*) rules. They have been first introduced in [1057]. This work addresses the problem of learning ground preconditions and effects for safe model free planning, a planning problem whose input are state atomic variables, initial and final state, and traces. This work has been extended to the problem of learning lifted preconditions and effects in the case of fluents (Boolean state variables) in [563]. Further extensions can deal with probabilities [562], with numeric action models [808], and in a multi agent setting [807]. The aforementioned works address the problem of learning action specificatons in the case of full observability.

Further offline approaches address the problem with different assumptions on the observability of states and actions. ARMS [1200] learns STRIPS action models from a set of successfully observed plans, without observing intermediate states, and making use of a MAX-SAT solver. SLAF [37] learns action models with universal quantifiers in effects, with partial observability of states, and making use of a SAT solver. LAMP [1232] learns action models with quantifiers and logical implication, under the hypothesis of partial observability, and using Markov logic networks. In [814], the authors learn action models with noisy and incomplete observations of states, and from successful and failing executions of actions. AMAN [1233] learns STRIPS action models from plan traces without observing states and from plans whose actions have a probability of being observed incorrectly (noisy actions). The work in [844] proposes a genetic algorithm to learn macro-actions (with negative preconditions) given a domain and action sequences. LOCM2 [262] learns from action sequences without any information about states, neither the initial nor the final state. The approach does not require to know the predicates of the planning domain. The FAMA system [18] learns STRIPS action models from examples by transforming the learning task into a classical planning task. The approach works with different kinds of inputs, from a set of plans to just a pair of initial and final states, without intermediate actions or states. Moreover, it accepts in input partially specified action models. The also provides an extensive and detailed comparison and classification of state of the art approaches to offline action model learning. The work in [672] proposes a technique based on probabilistic inference to learn action specifications from plan traces that are obtained by observing the environment states through noisy sensors.

The works in [164, 954] provide a framework for learning first-order symbolic representations from plain graphs, i.e., state transition systems generated by the execution of plans. While most of the works in offline learning action models assume that the actor gets input traces in the appropriate symbolic representation, a major distinguishing characteristic of this work is that the action model is learned from non-symbolic data, such as graphs representing the state transition system generated by the applications of actions. Moreover, the authors do not assume knowledge of the action schemas, predicate symbols, or objects. In particular, they learn action specifications that produce state-space graphs isomorphic to the input ones, by encoding the learning problem as a SAT problem. While in [164] the input graphs are assumed to be complete and without noise, in [954] these assumptions are relaxed by exploiting a more efficient encoding of the learning problem in answer set programming. Graph Neural Networks are exploited as a solver to learn generalized policies in [1053]. The work in [15] provides a general framework that, given in input a state transitions system, is able to synthesize different target languages for action specification, such as the synthesis of STRIPS action models, or the update rule of a cellular automaton. Given a set of examples of state-transitions, represented as (pre-state, action, post-state) tuples, the actor synthesizes a structured program that, when executed on a given pre-state, outputs its associated post-state. The synthesis method implements a combinatorial search in the space of well-structured terminating programs that can be built using a Random-Access Machine (RAM). In [1077], the authors propose an approach to learn type-generalized actions that can transfer to a variety of different and unknown situations and entities.

There are also works addressing the problem of learning action specifications by using natural language processing. For instance, the work in [14] proposes to use a neurosymbolic approach based on Logical Neural Networks, where neurons have meanings in weighted real-valued logic, to learn lifted logical operator models in PDDL through neuro-symbolic Inductive Logic Programming.

Most of the aforementioned approaches relax the assumption of full observability, they can deal with state variables whose value may be not accessible (hidden) in some states, and some of them deal even with noisy states and noisy actions. All of them assume that the actions to be executed is given in input with plan traces, and therefore do not deal with the problem of guiding the exploration phase towards informative applications of actions.

Online approaches

Since the seminal work on online learning of operators [421, 1153], and the first approaches to learning action models by integrating learning, planning, and execution [382], recent approaches have addressed the problem of online and incremental learning of action models. 3SG [211] is an online algorithm that learns probabilistic action models with conditional effects and deals with action failures, sensory noise, and incomplete information.

In [1191], the authors describe an instance-based online method for learning action models in relational domains. The work is extended to deal with both discrete and continuous action models in [1192, 1193]. The works in [952, 951] propose a technique based on relational reinforcement learning to learn deterministic action models, and [953] extends the approach to deal with nondeterministic actions.

OLAM [674] learns lifted action models (expressed in PDDL) under the the assumption of perfect (non-noisy) full observability of actions and the states reached by the agent. OLAM learns action models online, incrementally during the execution of plans, by combining and interleaving the activity of learning action preconditions and effects with an exploration phase that selects which plan to execute. Its main distinguishing characteristic is the ability to generate online informative traces, an important advantage w.r.t. all the offline approaches. OLAM generates informative traces by searching for informative states. Indeed, the idea of informative state-action pair presented in Section 4.2.2 has been inspired and adapted from [674]. While in Section 4.2.2 we deal with state variables, the work in [674] is limited to Boolean state variables. The OLAM online learning algorithm has been proved to be correct and complete for reachable states.

The work in [443] addresses the problem of repairing action specifications that are incomplete or incorrect. It uses automated planning to repair errors in the specification of actions that render the planning task unsolvable. This work focuses on missing action effects, which can compromise the task's solvability.

The work on planning by reinforcement learning (RL) [1070] (see Chapter 10), shares some similarities with the online approaches to learning action models, since both approaches learn action models online by applying actions in a simulated environment or by actually acting in a real world environment. However, both model based and model free RL focuses on learning policies for probabilistic models rather than action models for deterministic domains. Moreover, RL generates policies for state transition systems, where states and actions are atomic and ground. The work on action model learning deals with the different problem to learn lifted preconditions and effects, that can define the behaviour of actions in general, in different states. Finally, the work presented in this chapter does not require the definition of a reward function, a task that can be difficult and not natural in some cases.

Learning actions from continuous perceptions

The approach described in this Chapter is based on the assumption that perceptions are mapped directly into the value of state variables. In both the offline and the online approach, we assume that the actor gets the results of the application of an action in the appropriate symbolic representation.

However, in many applications, there is a huge gap between real perceptions and the symbolic abstract representation in state variables. Most often, an actor perceives the world and acts in it through sensors and actuators that work with data in a continuous space, typically represented with variables on real numbers.[8] For instance, a robot does not perceive directly the fact that it is in a given room/state, instead it perceives, e.g., to be in a position of the building through sensors like odometers or images from its RGB camera.

It is part of the cognitive capability of the actor to fill the gap between these two different levels of abstractions. For this reason, it is important to study and devise approaches that address the problem of learning how to map perceptions and observations represented with continuous variables into abstract models, in our case abstract deterministic symbolic models. While the problem of designing and implementing a mapping from continuous variables representing perceptions to abstract representations has been studied extensively, the problem of learning an abstract representation from continuous perceptions, as well the problem of learning the mapping between continuous perceptions and abstract representations, is far from obvious and it is deserving more and more research. For this reason, in the following, we discuss some recent and different approaches to learning deterministic models from continuous perceptions in the environment.

Causal InfoGAN [657] learns discrete or continuous models from high dimensional sequential observations with the objective to generate an execution trace in the high dimensional space. LatPlan [58, 57] takes in input pairs of high dimensional raw data (e.g., images) corresponding to transitions. It takes an offline approach. In a first phase, a State Autoencoder learns a mapping between raw data and abstract states, represented as vectors of binary state variables. In the second phase, LatPlan learns a transition function from the state pairs obtained by applying the mapping learned in the first phase to the training pairs. Planning is finally applied to the learned model. LatPlan has been shown experimentally to work with high dimensional data like images. In [714], the authors propose a framework that learns action models from parsed images given in a language used to describe 2D objects. The approach does not require to know the predicates of the planning domain.

In [637], STRIPS models are constructed by learning the Boolean atoms of the preconditions and effects of actions. The basic assumption is that a continuous model of the world is available, and that it is possible to know a fixed set-theoretic mapping from the continuous model to the deterministic classical planning domain.

[8]Even [164, 954], which try to learn action specification from execution graphs, do not deal with perceptions in a continuous space.

PAL [996] is based on a framework to learn a deterministic state transition system from observations of continuous variables through a perception function estimating the likelihood of being in a given state of the transition system. In [673], the idea is extended with a PDDL-based deterministic symbolic model that guides the exploration of the environment to learn the state transition system online and to scale up to large state spaces. OGAMUS [675] learns online the grounding of PDDL deterministic models by exploring unknown environments, mapping sensory data into symbolic states, and extending the signature of the symbolic model with new constants representing new objects discovered online in the environment. In [198], a state transition system is learned incrementally in an unknown environment. The learned model is reused for tackling the object goal navigation task. Each state is an abstraction of perceptions from high-dimensional sensory data (e.g., RGB-D images). A "planning for learning" approach (see Figure 1.2) is proposed in [676], where symbolic planning in a PDDL representation is used to train automatically a neural network for learning object properties by continuously collectomg training data obtained by exploring the environment. This work is extended in [677] by learning the PDDL preconditions under which the agent can perceive correctly an object property. The quality of the prediction of a deep neural network is evaluated by identifying, via clustering, which are the circumstances in which the predictions are correct with a certain level of confidence. In [1183], an end-to-end framework learns probabilistic state predictions from sequences of image-action pairs and infers lifted action schema.

A complementary approach is pursued in works that plan and learn directly in a continuous space, see e.g., [5], [797], [243]. These approaches do not require an abstract discrete model of the world. Such approaches are very suited to address some tasks, e.g., moving a robot arm to a desired position or performing some manipulations. However, in several situations, it is conceptually appropriate and practically efficient to learn an abstract discrete and deterministic model where planning is much easier and efficient to perform.

Approaches based on Large Language Models (LLMs) (see Chapter 23) constitute a potential new trend that in the future could be related to learning action specifications (see, e.g, [865])

The issues dealt by the above mentioned works on learning actions from continuous perceptions have also been addresses extensively and in depth by the robotics community, which has addressed the general problem of dealing with actuators that have to perform actions and sensors that perform perceptions in the real world (see, Part VII).

4.4 Exercises

4.1. Refine and implement the schema presented in Section 4.1.1, Algorithm 4.2.

4.2. Given a set of transitions (s, a, s'), and a relaxed plan heuristic h_0 how would you train a neural network to learn a better heuristic?

4.3. Rewrite the algorithms in Section 4.2.1 and Section 4.2.2 in the case all state variables are Boolean.

4.4. How would you extend the algorithms in Section 4.2.1 and Section 4.2.2 in the case some of the state variable values are hidden in some states?

4.5. How would you extend the algorithms in Section 4.2.1 and Section 4.2.2 in the case some of the state variable values are noisy, i.e., they do not provide a correct value for sensors?

Part II

Hierarchical Task Networks

Though this be madness, yet there is method in't.

William Shakespeare, *Hamlet*, circa 1600

Hierarchical Task Network (HTN) planning and acting operates at multiple levels of abstraction. Given a network of *tasks* that are activities to perform, the actor or planner refines[9] them into smaller and smaller tasks, proceeding until it finds executable actions. For the simple DWR domain in Figure 2.2, the following figure shows one way this might be done:

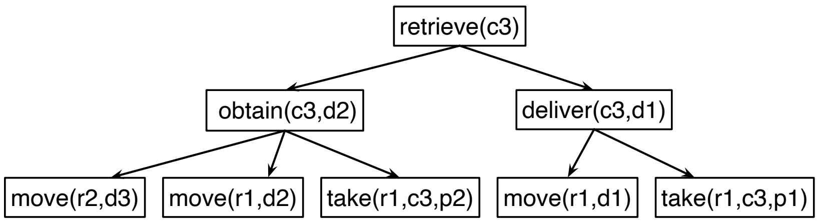


Figure II.1. Refinement of tasks into smaller tasks.

The refinement process is guided by *HTN methods*, each of which specifies a way to refine a task into subtasks. Some tasks may have several applicable methods, each of which proposes a different refinement, in which case the actor or planner may need to try several different refinements to find the best one for the problem at hand.

By specifying standard ways to perform tasks, HTN methods can implement not just the end-state constraints used in classical planning, but also constraints on a plan's trajectory that are difficult to encode as classical actions—as may occur in batch recipes, medical procedures, standard operating procedures, and the like. Furthermore, by focusing the search on specific ways to solve a problem, HTN methods can reduce the size of a planner's or actor's search space.

[9]Most of the HTN planning literature calls this *decomposing* the tasks, but we call it *refining* for consistency with the rest of this book.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

In complicated applications, significant effort may be needed to ensure that the HTN methods are correct and complete. Of course, similar effort may be needed to develop classical representations of complicated application domains [481].

This part is organized as follows. Chapter 5 is about representing HTN methods and using them for planning. Chapter 6 describes a reactive HTN acting procedure, some ways for an actor to use HTN planning algorithms, and some ways to recover when problems occur during plan execution. Chapter 7 describes some algorithms for learning HTN methods from example plan traces.

Later, in Parts VI and V of the book, some of the HTN concepts will be generalized to represent and reason about probabilistic action outcomes or temporal durations.

5 HTN Representation and Planning

This chapter is about representing HTN planning domains and solving HTN planning problems. Because HTN representation formalisms add HTN tasks and methods to classical domain models, several of the formal definitions require the same restrictions as in Part I. Most practical HTN implementations, however, loosen or drop several of these restrictions, such as the ones discussed in Remark 2.6.

This chapter is organized as follows. Section 5.1 is about ways to represent and solve planning problems in which there is a totally ordered sequence of tasks to accomplish. Section 5.2 generalizes these to allow partially ordered tasks. Section 5.3 describes ways to combine classical planning and HTN planning. Section 5.4 briefly discusses heuristic functions, expressivity, and computational complexity.

5.1 Totally Ordered Tasks

This section is about *total-order HTN* planning, which deals with totally ordered sequences of tasks. We will use the definitions of action schemas, object variables, and goal formulas in Chapter 3, and add definitions of tasks and methods.

Definition 5.1. A *task* is any of the following syntactic entities:

1. A *primitive task* is an instance (either ground or unground) of an action schema.
2. A *compound task* is a term of the form $\mathsf{name}(z_1, \ldots, z_k)$, where name is a symbol called the task's name, and each z_i is an object or an object variable.
3. A *goal task* is a classical goal formula, that is, a set of literals.

Compound tasks and goal tasks are also called *nonprimitive tasks*. □

Definition 5.2. A *total-order HTN method* is a tuple

$$m = (\mathrm{head}(m), \mathrm{task}(m), \mathrm{pre}(m), \mathrm{subtasks}(m)) \tag{5.1}$$

where:

- $\mathrm{head}(m)$ is a syntactic expression of the form $\mathsf{name}(z_1, \ldots, z_k)$, where name is a symbol called m's *name* and $(z_1, \ldots, z_k)$ is a list of zero or more *parameters*.
- $\mathrm{task}(m)$ is a nonprimitive task. This is the task for which m is *relevant*, and depending on its type, m is either a *compound-task method* or *goal method*.
- $\mathrm{pre}(m)$ is a set of zero or more literals that are called m's *preconditions*.
- $\mathrm{subtasks}(m)$ is a sequence of zero or more tasks that are called m's *subtasks*.
- If m is a goal method, then to ensure that m will accomplish $\mathrm{task}(m)$, the last element of $\mathrm{subtasks}(m)$ must be either $\mathrm{task}(m)$ or a primitive task α such that $\mathrm{eff}(\alpha) \models \mathrm{task}(m)$.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- The parameters in head(m) are not required to be object variables; they may also be object constants and state variables.[1] However, every object variable that occurs anywhere in m must also occur somewhere in head(m). □

Notation and terminology: Rather than writing methods as tuples, we usually will use the following pseudocode format:

```
$\mathit{name}(z_1, z_2, \ldots, z_k)$
  task: $t$
  pre: $p_1, \ldots, p_m$
  sub: $t_1, \ldots, t_n$
```

which says that head(m) = $\mathit{name}(z_1, \ldots, z_n)$, task$(m) = t$, pre$(m) = \{p_1, \ldots, p_m\}$, and subtasks$(m) = \{t_1, \ldots, t_n\}$.

Example 5.3. Let Σ be the classical planning domain in Example 2.1, and consider the goal task $\{\mathsf{pile}(c) = p\}$. Here is a method for this task. Its parameters are $r \in \mathit{Robots}$; $d \in \mathit{Docks}$; $c, c' \in \mathit{Containers}$; and $p \in \mathit{Piles}$:

```
m1-put-in-pile$(r, c, p, d)$
  task: $\{\mathsf{pile}(c) = p\}$
  pre: $\mathsf{at}(p, d), \mathsf{pile}(c) \neq p, \mathsf{cargo}(r) = \mathsf{nil}$
  sub: $\mathsf{get\text{-}container}(r, c), \mathsf{navigate}(r, d), \mathsf{put}(r, c, \mathsf{top}(p), p, d)$
```

The preconditions require that pile p is at loading dock d, container c isn't already part of p, and robot r isn't carrying anything.

The first two tasks in the subtask list are compound, and Example 5.4 will give methods for them. The last subtask is an instance[2] of the put action in Example 2.8, and its effects include $\mathsf{pile}(c) = p$. This satisfies the requirement in Definition 5.2 for the last subtask of a goal method. □

5.1.1 Total-Order HTN Planning Domains

A *total-order HTN planning domain* is a tuple

$$\Sigma = (\Sigma_c, \mathcal{M}), \tag{5.2}$$

where Σ_c is a classical planning domain in state-variable representation, and $\mathcal{M}$ is a set of total-order HTN methods subject to the following restrictions: every $m \in \mathcal{M}$ has a unique name, every parameter of m is an object variable, and every argument of task(m) is an object variable.

Because every object variable in m is a parameter of m, it follows that every instance of m can be unambiguously identified by its head. Thus when referring to an instance of m, we will usually will write just its head rather than the entire method.

[1]This is to allow m to be an instance of another method. Later, Equation 5.2 will require that in HTN domain definitions, the methods' parameters all are variables.

[2]Some HTN formalisms would not allow $\mathsf{top}(p)$ to appear in the subtask's argument list. To satisfy such a restriction, we can replace $\mathsf{top}(p)$ with a new variable c', and give the method an additional precondition $c' = \mathsf{top}(p)$. Similar changes can be made to the other methods in this chapter.

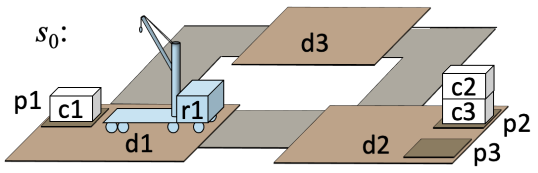


Figure 5.1. The state s_0 in Equation 5.6.

We let

$$Ground(\mathcal{M}) = \{\text{all ground instances of methods in } \mathcal{M}\}. \tag{5.3}$$

A ground method m is *applicable* in a state s if $s \models \text{pre}(m)$. Furthermore, m is *relevant* for a task t if either t is a compound task and $\text{task}(m) = t$, or t is a goal task and $\text{task}(m) \models t$. We let

$$Methods(s, t, \mathcal{M}) = \{m \in Ground(\mathcal{M}) \mid m \text{ is applicable in } s \text{ and relevant for } t\}. \tag{5.4}$$

If t is a goal task, then

$$Actions(s, t) = \{a \in Applicable(s) \mid \gamma(s, a) \models t\}. \tag{5.5}$$

Example 5.4. Let Σ_c be a classical planning domain in which the objects, rigid relations, and states are the same as in Example 2.1 except that there is only one robot, r1, and the actions are the ground instances of the action schemas in Example 2.8. Figure 5.1 shows the following state:

$$\begin{aligned} s_0 = \{&\mathsf{cargo(r1) = nil}, && \mathsf{loc(r1) = d1}, && \\ &\mathsf{occupied(d1) = T}, && \mathsf{occupied(d2) = F}, && \mathsf{occupied(d3) = F}, \\ &\mathsf{pile(c1) = p1}, && \mathsf{pile(c2) = p2}, && \mathsf{pile(c3) = p2}, \\ &\mathsf{pos(c1) = nil}, && \mathsf{pos(c2) = c3}, && \mathsf{pos(c3) = nil}, \\ &\mathsf{top(p1) = c1}, && \mathsf{top(p2) = c2}, && \mathsf{top(p3) = nil}\}. \end{aligned} \tag{5.6}$$

The total-order HTN planning domain is $\Sigma = (\Sigma_c, \mathcal{M})$, where $\mathcal{M}$ contains eight methods. They have the following parameters with the following ranges: $r \in \mathit{Robots}$; $c \in \mathit{Containers}$; $d, d' \in \mathit{Docks}$; $p, p' \in \mathit{Piles}$.

The first method in $\mathcal{M}$ is m1-put-in-pile from Example 5.3. Next are two methods for m1-put-in-pile's subtask get-container:

m1-get-container(r, c)	m2-get-container(r, c, p, d)
task: get-container(r, c)	task: get-container(r, c)
pre: cargo$(r) = c$	pre: cargo$(r) =$ nil, pile$(c) = p$, at(p, d)
sub: *// no subtasks*	sub: navigate(r, d), uncover(c), take$(r, c,$ pos$(c), p, d)$

The method m1-get-container is for the case where r is already carrying c and thus nothing needs to be done. In m2-get-container, the first precondition makes it applicable only if r is not carrying anything, and its other two preconditions ensure that p and d have the correct values. Its uncover and navigate subtasks are described in the following paragraphs, and the take is one of the actions in Example 2.8. There are no methods for cases where r is carrying something other than c.

Next are two methods for uncover(c), the task of removing all containers above c:

```
m1-uncover(c)
  task: uncover(c)
  pre:  top(pile(c)) = c
  sub:  // no subtasks

m2-uncover(r, c, p, c', p', d)
  task: uncover(c)
  pre:  pile(c) = p, top(p) = c', c' ≠ c,     (1)
        at(p, d), at(p', d), p ≠ p',           (2)
        loc(r) = d, cargo(r) = nil             (3)
  sub:  take(r, c', pos(c'), p, d),
        put(r, c', top(p'), p', d),
        uncover(c)
```

The method m1-uncover is for the case where nothing needs to be done because c is at the top of its pile. In m2-uncover, the three lines of preconditions require that (1) c is in a pile p but not at the top of p, (2) both p and another pile p' are at the same loading dock d, and (3) r is at d and isn't carrying anything. The subtasks are to move r to d, remove the topmost container above c, and call the task uncover(c) recursively. The recursive calls will remove the rest of the containers above c.

Finally, there are three methods for navigate(r, d), the task of moving r to d. We include them for illustrative purposes, but we do not recommend using them unless the planning domain is quite small, because they can produce a huge search space. A domain-specific heuristic function could be used to avoid most of the search space, but in most practical applications one would instead use a route-planning algorithm. Here are the three methods:

```
m1-navigate(r, d)
  task: navigate(r, d)
  pre:  loc(r) = d
  sub:  // no subtasks

m2-navigate(r, d', d)
  task: navigate(r, d)
  pre:  adjacent(d', d), loc(r) = d'
  sub:  move(r, d', d)

m3-navigate(r, d', d)
  task: navigate(r, d)
  pre:  loc(r) ≠ d, ¬adjacent(loc(r), d), adjacent(loc(r), d')
  sub:  move(r, loc(r), d'),          // primitive task
        navigate(r, d)                // compound task
```

If r is already at d, m1-navigate is applicable and does nothing. If r's location is adjacent to d, m2-navigate moves r to d using the action move in Example 2.8. If r's location is not adjacent to d, then m3-navigate moves r to another dock d' and calls navigate recursively to try to get from d' to d.

Thus the methods in $\mathcal{M}$ are m1-put-in-pile, m1-get-container, m2-get-container, m1-uncover, m2-uncover, m1-navigate, m2-navigate, and m3-navigate. □

If $\Sigma = (\Sigma_c, \mathcal{M})$ is a total-order HTN planning domain and $(O, R, X, \mathcal{A})$ is the state-variable representation of Σ_c, then the tasks in Σ include task(m) for every ground method m in $\mathcal{M}$, every set of literals in Σ_c, and every instance of the action schemas in $\mathcal{A}$.

Example 5.5. In Example 5.4, all instances of put-in-pile(c, p), uncover(c), and navigate(r, d) are compound tasks. The goal tasks include all sets of literals in Σ_c, hence include m1-put-in-pile's goal task $\{\text{pile}(c) = p\}$. All instances of take(r, c, c', p, d), put(r, c, c', p, d), and move(r, d, d') are primitive tasks. □

5.1.2 Total-Order HTN Planning Problems

Definition 5.6. A *total-order HTN planning problem* is a tuple $P = (\Sigma, s_0, T)$, where $\Sigma = (\Sigma_c, \mathcal{M})$ is a total-order HTN planning domain, s_0 is P's initial state, and T is a sequence of ground tasks.

Solutions for P are defined inductively as follows. If T is empty, then the empty plan $\langle\rangle$ is a solution for P. Otherwise, let t be the first task of T, so that $T = t \cdot T'$ where T' is a (possibly empty) sequence of tasks. Then:

1. If t is an action in $\mathit{Applicable}(s_0)$, then for every solution π for the problem $(\Sigma, \gamma(s_0, t), T')$, the plan $t \cdot \pi$ is a solution for P.
2. If t is a compound task or goal task and $m \in \mathit{Methods}(s_0, t, \mathcal{M})$, then every solution for the problem $(\Sigma, s_0, \text{subtasks}(m) \cdot T')$ is also a solution for P.
3. If t is a goal task and $a \in \mathit{Actions}(s_0, t)$, then[3] for every solution π for the problem $(\Sigma, \gamma(s_0, a), T')$, the plan $a \cdot \pi$ is a solution for P.
4. If t is a goal task and $s_0 \models t$, then every solution for the problem (Σ, s_0, T') is also a solution for P. □

Example 5.7. Let Σ and s_0 be as in Example 5.4, and suppose we want to move container c1 to pile p2. The goal task is $\{\text{pile(c1)} = \text{p2}\}$, so the planning problem is

$$P = (\Sigma, s_0, \langle\{\text{pile(c1)} = \text{p2}\}\rangle), \tag{5.7}$$

which has one solution:

$$\pi = \langle \text{take(r1, c1, c2, p1, d1)},\ \text{move(r1, d1, d2)},\ \text{put(r1, c1, c3, p2, d2)} \rangle.$$

Figure 5.2 is a refinement tree (see next paragraph) that shows the derivation of π. □

Definition 5.8. Let π be a solution for a total-order HTN planning problem $P = (\Sigma, s_0, T)$. A *refinement tree* for π is a tree in which each node is a tuple

$$v = (\mathit{label}(v), \mathit{content}(v), \mathit{parent}(v), \mathit{Children}(v)),$$

where $\mathit{label}(v)$ is a unique identifier, $\mathit{content}(v)$ is a ground task or ground method, $\mathit{parent}(v)$ is v's parent, and $\mathit{Children}(v)$ is a sequence of children. As a special case, the root node has $\mathit{content}(v) = \text{root}$ and $\mathit{parent}(v) = \text{nil}$.

The nodes are organized as follows:

- *Root node.* If $\mathit{content}(v) = \text{root}$, then for each t_i in T, v has a child v_i with $\mathit{content}(v_i) = t_i$. Notice that if T is empty then v has no children.

[3]The requirement that $a \in \mathit{Actions}(s_0, t)$ prevents arbitrary action sequences from being solutions, unlike in classical planning.

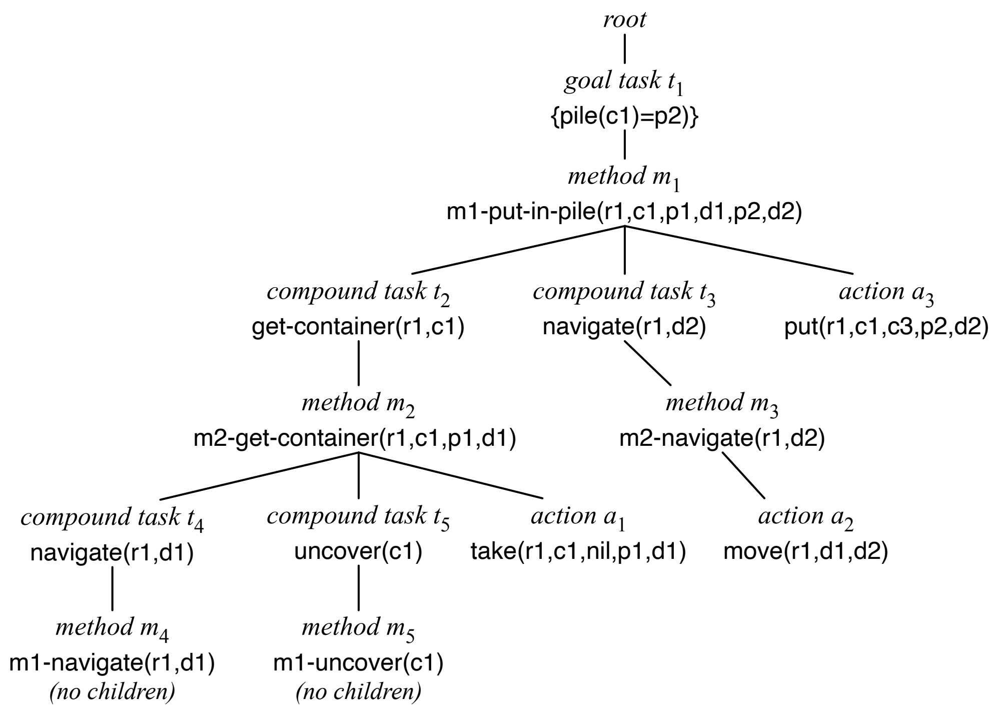


Figure 5.2. A refinement tree (see Definition 5.8) for the plan π in Example 5.7. At each node, the first line gives the kind of node and its label, and the second line is the node's content.

- *Action nodes.* If $content(v)$ is a primitive task (hence an action, because v is ground), then v has no children.
- *Compound-task nodes.* If $content(v)$ is a compound task and m is the ground methods that refined it, then v has one child v', with $content(v') = m$.
- *Goal-task nodes.* If $content(v)$ is a goal task and m is the ground method or action that refined it, then v has one child v', with $content(v') = m$. If $content(v)$ wasn't refined because it was already true in the current state, then v has no children.
- *Method nodes.* If $content(v)$ is a ground method m, then for each task t_i in subtasks(m), v has a child v_i with $content(v_i) = t_i$. Note that if subtasks(m) is empty then v has no children.

If $\pi = \langle a_1, \ldots, a_n \rangle$, then $a_1, \ldots, a_n$ will be the contents of the tree's action nodes in left-to-right order. □

5.1.3 Total-Order HTN Planning Algorithms

Algorithm 5.1, TO-HTN-Forward, is a nondeterministic total-order HTN planning algorithm based on Definition 5.6. In Line 1, it calls the subroutine HTN-Get-Candidates to get a set M of candidate ground methods and actions for the first

```
TO-HTN-Forward($\Sigma_c, \mathcal{M}, s, T$)
    if $T$ is empty then return $\langle\rangle$
    $t \leftarrow$ the first element of $T$;  $T' \leftarrow$ the rest of $T$
1   $M \leftarrow$ HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
    if $M = \varnothing$ then return failure
    nondeterministically choose $m \in M$
    switch $m$ do
2       case $m$ is an action do
            $\pi \leftarrow$ TO-HTN-Forward($\Sigma_c, \mathcal{M}, \gamma(s, m), T'$)
            if $\pi \neq$ failure then return $m \cdot \pi$
            else return failure
3       case $m$ is a ground method do
            return TO-HTN-Forward($\Sigma_c, \mathcal{M}, s, \text{subtasks}(m) \cdot T'$)

HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
    switch $t$ do
4       case $t$ is an action do
            if $t$ is applicable in $s$ then $M \leftarrow \{a\}$
            else $M \leftarrow \varnothing$
5       case $t$ is a compound task do $M \leftarrow Methods(s, t, \mathcal{M})$
6       case $t$ is a goal task do
            $M \leftarrow Methods(s, t, \mathcal{M}) \cup Actions(s, t)$
7           if $s \models t$ then $M \leftarrow M \cup \{\text{null}\}$
    return $\mathcal{M}$
```

Algorithm 5.1. TO-HTN-Forward, which plans for totally-ordered tasks.

element t of T. The subroutine's cases correspond to the *if* parts of Definition 5.6's numbered clauses:

- Line 4 corresponds to clause 1, in which t is an action. If t is applicable in s, then it is put into M.
- Line 5 corresponds to clause 2, in which task(τ) is a compound task. In this case, the ground methods that are both relevant and applicable are put into M.
- Line 6 corresponds to clauses 2, 3, and 4, in which task(τ) is a compound task. For each applicable clause, the candidates it produces are put into M.
- Line 5 corresponds to item 4, in which t is already true. To implement this case, null is a dummy method with no preconditions and no subtasks.[4]

Once it has M, TO-HTN-Forward nondeterministically chooses an element of M and

[4]Line 7 is written to ensure that when $s \models t$, TO-HTN-Forward will consider both empty and nonempty plans for achieving t, because a nonempty plan might have side-effects that are needed later in T. However, if the HTN methods in Σ are well-written then such situations should not occur. In this case, TO-HTN-Forward's search space can be reduced by changing Line 7 to $M \leftarrow \{\text{null}\}$.

calls itself recursively in lines 2 and 3, which correspond to the "then" parts of Definition 5.6's numbered clauses.

TO-HTN-Forward can be proved sound and complete by induction (see Exercise 5.5). If (Σ, s, T) is a solvable total-order HTN planning problem and π is the empty plan, then at least one of TO-HTN-Forward's nondeterministic traces will return a solution plan.

Algorithm 5.2, TO-HTN-Forward-Det, is a deterministic version of TO-HTN-Forward. Its *Frontier*, *Children*, and *Expanded* sets are like the ones in Forward-Search-Det, but each node is a triple (π, s, T) in which π is a plan, $s = \gamma(s_0, \pi)$, and T is a sequence of tasks. Line 2 produces the same effect as line 7 of TO-HTN-Forward.

Algorithm 5.3, TO-HTN-Forward-RT, is a modified version of TO-HTN-Forward that makes a subtree for each $t \in T$ and returns a refinement tree like the one in Figure 5.2. It should be called with *parent* = nil. Its subroutine RT-Make-Node is for making new nodes and adding them to the tree. In Line 1, the *if* test excludes special cases in which a new node isn't needed.

```
TO-HTN-Forward-Det($\Sigma_c, \mathcal{M}, s_0, T_0$)
    $Frontier \leftarrow \{(\langle\rangle, s_0, T_0)\}$    // {initial node}
    $Expanded \leftarrow \varnothing$
    while $Frontier \neq \varnothing$ do
1       select a node $v = (\pi, s, T) \in Frontier$
        remove $v$ from $Frontier$ and add it to $Expanded$
        if $T = \langle\rangle$ then return $\pi$
        $t \leftarrow$ the first element of $T$;  $T' \leftarrow$ the rest of $T$
        switch $t$ do
            case $t$ is an action do
                if $t$ is applicable in $s$ then $Children \leftarrow \{(\pi \cdot t, \gamma(s, t), T')\}$
                else $Children \leftarrow \varnothing$
            case $t$ is a compound task do
                $Children \leftarrow \{(\pi, s, \text{subtasks}(m) \cdot T') \mid m \in Methods(s, t, \mathcal{M})\}$
            case $t$ is a goal task do
                $Children \leftarrow \{(\pi \cdot a, \gamma(s, a), T') \mid a \in Actions(s, t)\}$
                        $\cup \{(\pi, s, \text{subtasks}(m) \cdot T') \mid m \in Methods(s, t, \mathcal{M})\}$
2               if $s \models t$ then $Children \leftarrow Children \cup \{(\pi, s, T')\}$
        prune 0 or more nodes from $Children$, $Frontier$ and $Expanded$
        $Frontier \leftarrow Frontier \cup Children$
    return failure
```

Algorithm 5.2. TO-HTN-Forward-Det, a deterministic version of TO-HTN-Forward.

```
TO-HTN-Forward-RT($\Sigma_c, \mathcal{M}, s, T, parent$)
    if $parent$ = nil then $parent \leftarrow$ RT-Make-Node(root, nil)
    foreach $t \in T$ do
        $v \leftarrow$ RT-Make-Node($t, parent$)
        $M \leftarrow$ HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
        if $M = \varnothing$ then return failure
1       if $t$ is not an action and $t \neq$ null then $v \leftarrow$ RT-Make-Node($m, v$)
        nondeterministically choose $m \in M$
        if $m$ is a ground method then
            if TO-HTN-Forward-RT($\Sigma_c, \mathcal{M}, s$, subtasks($m$), $v$) = failure then
                return failure
        else if $m$ is an action then $s \leftarrow \gamma(s, t)$
    return $parent$

RT-Make-Node($task\text{-}or\text{-}method, parent$)
    $l \leftarrow$ a new label that depends on $task\text{-}or\text{-}method$
    $v \leftarrow$ a new node $(l, task\text{-}or\text{-}method, parent, \langle\rangle)$
    if $parent \neq$ nil then append $v$ to $Children(parent)$
    return $v$
```

Algorithm 5.3. TO-HTN-Forward-RT, which returns a refinement tree. It should be called with *parent* = nil. The subroutine HTN-Get-Candidates is the same as in TO-HTN-Forward.

5.1.4 Serially Solvable Planning Problems

A total-order HTN planner's search space can be greatly reduced if its planning problem $P = (\Sigma, s_0, T)$ is *serially solvable*, a condition that is defined inductively as follows. If $T = \langle\rangle$ then P is serially solvable. If $T \neq \langle\rangle$, then let t be the first element of T, so that $T = t \cdot T'$ for some T'. Then P is serially solvable if for every plan π that solves $(\Sigma, s_0, \langle t\rangle)$, the planning problem $(\Sigma, \gamma(s_0, \pi), T')$ is serially solvable.

Example 5.9. The planning problem in Example 5.4 is serially solvable. □

If $P = (\Sigma, s_0, \langle t_1, \ldots, t_n\rangle)$ is serially solvable, then whenever TO-HTN-Forward-Det finds a plan π_1 to accomplish t_1, it can prune all of the other paths in its search space, because P is guaranteed to have a solution that starts with π_1. Applying this argument repeatedly, each time it find plans for $t_2, \ldots, t_n$, it can prune all of the other paths in its search space. Algorithm 5.4, TO-HTN-Serial, is an algorithm that works this way.

If a planning problem is not serially solvable, a planner may still be able to prune large parts of its search space if parts of the problem are serially solvable.

5.2 Partially Ordered Tasks

Sometimes it is undesirable to specify a total order on a set of tasks. If several possible orderings are acceptable, then we might want to specify a partial ordering and let the planner decide which total ordering to use. To represent partially ordered tasks, we define a *partially ordered task network* to be a pair

$$\mathcal{T} = (T, \prec), \tag{5.8}$$

where T is a set of *task nodes* and $\prec$ is a partial ordering of T. Each task node is a pair $\tau = (\text{label}(\tau), \text{task}(\tau))$, where $\text{task}(\tau)$ is a task and $\text{label}(\tau)$ is a unique identifier. Thus a task may occur in T more than once with different labels.

In this section, “task network” will mean a partially ordered task network.

A *partial-order HTN method* is a tuple

$$m = (\text{head}(m), \text{task}(m), \text{pre}(m), \text{subtasks}(m), \prec_m), \tag{5.9}$$

where $\text{head}(m)$, $\text{task}(m)$, and $\text{pre}(m)$ are as in Equation 5.1, and $(\text{subtasks}(m), \prec_m)$ is a task network. We normally will write partial-order HTN methods as pseudocode instead of tuples; Example 5.10 will give several examples.

5.2.1 Planning Domains and Problems

A *partial-order HTN planning domain* is a pair

$$\Sigma = (\Sigma_c, \mathcal{M}), \tag{5.10}$$

```
TO-HTN-Serial($\Sigma_c, \mathcal{M}, s, T$)
    $\pi \leftarrow \langle\rangle$
    while $T \neq \langle\rangle$ do
        $t \leftarrow$ the first element of $T$;  $T \leftarrow$ the rest of $T$
        if $t$ is an action then
            if $t$ is not applicable in $s$ then return failure
            $s \leftarrow \gamma(s,t)$;  $\pi \leftarrow \pi \cdot t$
        else if $t$ is a goal task and $Actions(s,t) \neq \varnothing$ then
            arbitrarily select $a \in Actions(s,t)$
            $s \leftarrow \gamma(s,a)$;  $\pi \leftarrow \pi \cdot a$
        else
            $M \leftarrow Methods(s,t,\mathcal{M})$
            if $M = \varnothing$ then return failure
            nondeterministically choose $m \in M$
            $\pi' \leftarrow$ TO-HTN-Serial($\Sigma_c, \mathcal{M}, s$, subtasks($m$))
            if $\pi'$ = failure then return failure
            $s \leftarrow \gamma(s,\pi')$;  $\pi \leftarrow \pi \cdot \pi'$
    return $\pi$
```

Algorithm 5.4. An algorithm for serially solvable total-order HTN planning problems.

where $\Sigma_c = (S, A, \gamma, \text{cost})$ is a classical planning domain and $\mathcal{M}$ is a set of partial-order HTN methods.

Total-order HTN planning can be viewed as a special case of partial-order HTN planning, because total-order HTN methods can be translated into equivalent partial-order HTN methods in linear time. For a total-order HTN method with subtasks $t_1, \ldots, t_n$, the corresponding partial-order HTN method has a label l_i for each t_i, with ordering constraints $l_1 \prec l_2 \prec \ldots \prec l_n$.

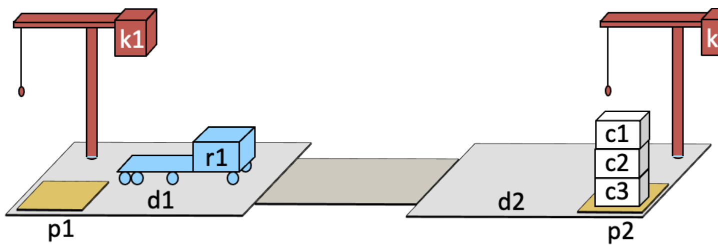


Figure 5.3. A DWR example in which cranes are at loading docks, not on robots.

Example 5.10. Let Σ_c be the DWR domain shown in Figure 5.3, in which the cranes are at loading docks, not on robots. The rigid relations and state variables in Σ_c are like the ones in Example 2.1, but with three differences. First, if k is a crane, then at(k, d) means that k is attached to loading dock d. Second, there is a new state variable holding(k) whose value is either a container c if k is holding c, or nil if k is empty. Third, the range of pos(c) is $\mathit{Cranes} \cup \mathit{Piles} \cup$ nil.

The action schemas for Σ_c include move from Example 2.8, and also the following:

```
unstack$(k, c, c', p, d)$        // take container c from pile p
    pre: at$(k,d)$, at$(p,d)$, holding$(k)$ = nil, pos$(c) = c'$, top$(p) = c$
    eff: holding$(k) \leftarrow c$, pos$(c) \leftarrow k$, pile$(c) \leftarrow$ nil, top$(p) \leftarrow c'$

stack$(k, c, c', p, d)$          // put container c onto pile p
    pre: at$(k,d)$, at$(p,d)$, holding$(k) = c$, top$(p) \leftarrow c'$
    eff: holding$(k) \leftarrow$ nil, pos$(c) = c'$, pile$(c) \leftarrow p$, top$(p) = c$

unload$(k, c, r, d)$             // take container c from robot r
    pre: at$(k,d)$, holding$(k) = c$, loc$(r) = d$
    eff: cargo$(r) \leftarrow c$, pos$(c) \leftarrow r$, holding$(k) \leftarrow$ nil

load$(k, c, r, d)$               // put container c onto robot r
    pre: at$(k,d)$, holding$(k)$ = nil, loc$(r) = d$, cargo$(r) = c$
    eff: pos$(c) \leftarrow k$, holding$(k) \leftarrow c$, cargo$(r) \leftarrow$ nil
```

In the partial-order HTN domain, put-on-robot(c, r) is the task of putting container c onto robot r. Here is a method to do this when r is empty, c is at the top of a pile p, and an empty crane is available. The method's partial-ordering constraints say that navigate and unstack may be done in either order, but both must be done before load.

```
m1-put-on-robot(k, c, c', r, d, p)
   task: put-on-robot(c, r)
    pre: cargo(r) = nil, top(p) = c, at(p, d),
         attached(k, d), holding(k) = nil
    sub: (t1, navigate(r, d)),            // compound task
         (t2, unstack(k, c, c', p, d)),   // action
         (t3, load(k, c, r, d))           // action
      ≺: t1 ≺ t3, t2 ≺ t3
```

The navigate task has two methods that are adaptations of the ones in Example 5.4:

```
m1-navigate(r, d)              m2-navigate(r, d', d)
   task: navigate(r, d)           task: navigate(r, d)
    pre: loc(r) = d                pre: adjacent(d', d), loc(r) = d'
    sub:    // none                sub: (t1, move(r, d', d))
      ≺:    // none                  ≺:    // none
```

The partial-order HTN planning domain is $\Sigma = (\Sigma_c, \mathcal{M})$, where $\mathcal{M} = \{$m1-put-on-robot, m1-navigate, m2-navigate$\}$. □

Here are some basic operations on ground task networks.[5] Let $\mathcal{T}_1 = (T_1, \prec_1)$ and $\mathcal{T}_2 = (T_2, \prec_2)$ be ground task networks that have no labels in common. Then:

- The *union* of $\mathcal{T}_1$ and $\mathcal{T}_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2 = (T_1 \cup T_2, \prec_1 \cup \prec_2)$.
- If τ is a task node in $\mathcal{T}_1$, the task network produced by *removing* τ from $\mathcal{T}_1$ is

$$\mathcal{T}_1 \setminus \{\tau\} = (T_1', \prec_1'), \tag{5.11}$$

  where $T_1' = T_1 \setminus \{\tau\}$, and $\prec_1'$ is the restriction of $\prec_1$ to T_1'.
- Let $m = (\text{head}(m), \text{task}(m), \text{pre}(m), \text{subtasks}(m), \prec_m)$ be a ground method that is relevant for τ. Then the task network produced from $\mathcal{T}_1$ by *refining* τ with m is

$$\mathit{refine}(\mathcal{T}_1, \tau, m) = (T_1' \cup \text{subtasks}(m),\ \prec_1' \cup \prec_\tau \cup \prec_m), \tag{5.12}$$

  where T_1' and $\prec_1'$ are as in Equation 5.11, $\prec_\tau$ is a partial ordering that constrains the nodes of T_1' that were before (or after) τ to be before (or after, respectively) the nodes of subtasks(m). Formally, for every $\tau_1 \in T_1'$ and $\tau_m \in \text{subtasks}(m)$, if $\tau_1 \prec_1 \tau$ then $\tau_1 \prec_\tau \tau_m$, and if $\tau \prec_1 \tau_1$ then $\tau_m \prec_\tau \tau_1$.

A *partial-order HTN planning problem* is a tuple

$$P = (\Sigma, s_0, \mathcal{T}), \tag{5.13}$$

where Σ is a partial-order HTN planning domain, s_0 is the initial state, and $\mathcal{T}$ is a ground task network.

Solutions to partial-order HTN planning problems can be defined in two ways. In the following definition, they are ordinary (totally ordered) plans. An alternative is to allow the plans to be partially ordered.

[5]These operations can be generalized to unground task networks and methods, by renaming object variables so that $\mathcal{T}_1$, $\mathcal{T}_2$, and m have no variable names in common.

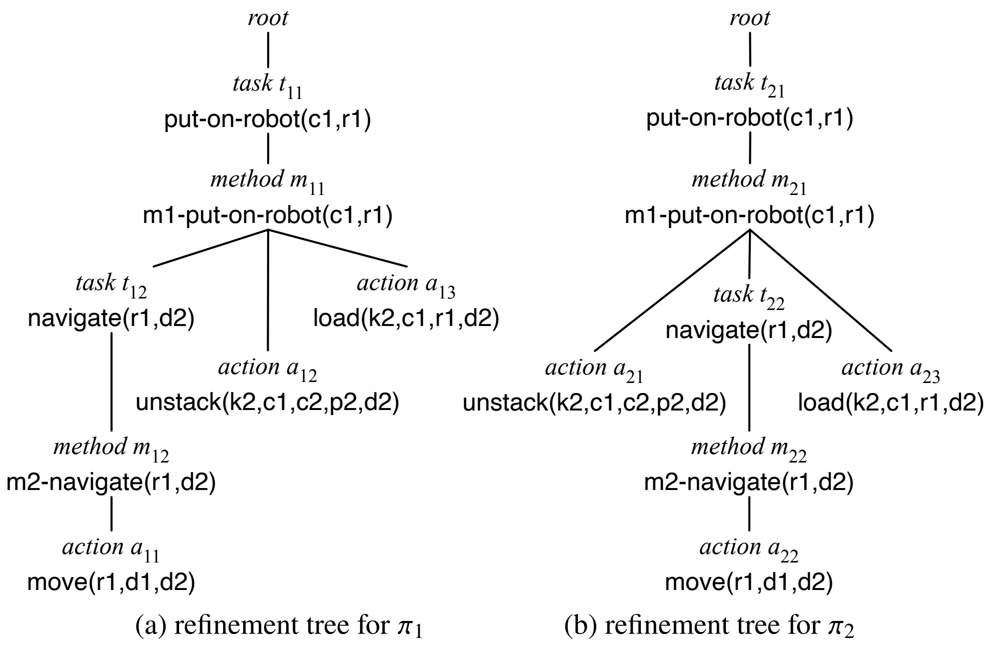


Figure 5.4. Refinement trees for Example 5.12. At each node, the first line tells what kind of node and gives its label, and the second line is the node's content.

Definition 5.11. A *solution* for a partial-order HTN planning problem $P = (\Sigma, s_0, \mathcal{T})$ is defined inductively as follows. If $\mathcal{T}$ is empty, then $\langle\rangle$ is a solution for P. If $\mathcal{T}$ is not empty, then let τ be any task node in $\mathcal{T}$ that has no predecessors in $\mathcal{T}$, let $t = \text{task}(\tau)$, and let $\mathcal{T}' = \mathcal{T} \setminus \{\tau\}$. Then:

1. If t is an action in $\mathit{Applicable}(s_0)$, then for every solution π for the problem $(\Sigma, \gamma(s_0, t), \mathcal{T}')$, the plan $t \cdot \pi$ is a solution for P.
2. If t is a goal task or compound task and $m \in \mathit{Methods}(s_0, t, \mathcal{M})$, then every solution for the problem $(\Sigma, s_0, \mathit{refine}(\mathcal{T}, \tau, m))$ is also a solution for P.
3. If t is a goal task and $a \in \mathit{Actions}(s, t)$, then for every solution π for the problem $(\Sigma, \gamma(s_0, a), \mathcal{T}')$, the plan $a \cdot \pi$ is a solution for P.
4. If t is a goal task and $s_0 \models t$, then every solution for the problem $(\Sigma, s_0, \mathcal{T}')$ is also a solution for P. □

If π is a solution for P, then the definition of a refinement tree for π is the same as Definition 5.8, with $\mathcal{T}$ substituted for T.

Example 5.12. Let Σ be as in Example 5.10, s_0 be as in Figure 5.3, and $\mathcal{T} = (T, \prec)$, where $T = \{\text{put-on-robot(c1,r1)}\}$ and $\prec$ is empty. Then $P = (\Sigma, s_0, \mathcal{T})$ has two solutions, which are identical except for the ordering of the first two actions:

$$\pi_1 = \text{move(r1,d1,d2), unstack(k2,c1,c2,p2,d2), load(k2,c1,r1,d2)},$$
$$\pi_2 = \text{unstack(k2,c1,c2,p2,d2), move(r1,d1,d2), load(k2,c1,r1,d2)}.$$

Figure 5.4 shows refinement trees for both of them. □

```
PO-HTN-Forward($\Sigma_c, \mathcal{M}, s, \mathcal{T}$)
    if $\mathcal{T}$ is empty then return $\langle\rangle$
1   nondeterministically choose a node $\tau$ in $\mathcal{T}$ that has no predecessors in $\mathcal{T}$
    foreach $\tau'$ in $\mathcal{T}$ that has no predecessors in $\mathcal{T}$ do
2       if $\tau' \neq \tau$ then add ordering constraints to $\mathcal{T}$ to make $\tau \prec \tau'$
    $t \leftarrow$ task($\tau$)
    $M \leftarrow$ HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
    if $M \neq \varnothing$ then
        nondeterministically choose $m \in M$
        if $m$ is an action then
            $\pi \leftarrow$ PO-HTN-Forward($\Sigma_c, \mathcal{M}, \gamma(s,a), \mathcal{T} \setminus \{\tau\}$)
            if $\pi \neq$ failure then return $a \cdot \pi$
        else if $m$ is a ground method then
            return PO-HTN-Forward($\Sigma_c, \mathcal{M}, s, refine(\mathcal{T}, \tau, m)$)
    return failure
```

Algorithm 5.5. A planning algorithm for partially-ordered tasks. The subroutine HTN-Get-Candidates is the same as in TO-HTN-Forward.

5.2.2 Partial-Order HTN Planning

Algorithm 5.5, PO-HTN-Forward, is a straightforward implementation of Definition 5.11. In the definition, $t = \text{task}(\tau)$ is always the first task in the solution. Lines 1–2 of the algorithm choose this task and ensure that it will come first. The rest of the algorithm uses Equations 5.11 and 5.12 for removal and refinement of nodes in $\mathcal{T}$, but otherwise is nearly identical to TO-HTN-Forward. A partial-order HTN version of TO-HTN-Forward-RT can also be written.

PO-HTN-Forward can be proved sound and complete by induction. If a partial-order HTN planning problem $(\Sigma, s, \mathcal{T})$ has at least one solution, then at least one of PO-HTN-Forward's nondeterministic traces will return a solution.

5.2.3 Plan-Space partial-order HTN Planning

We now describe a plan-space planning algorithm for partial-order HTN problems. It is identical to PSP, Algorithm 3.11, except that it has an additional parameter, the set of methods $\mathcal{M}$. However, there are changes to some of the definitions of the entities that PSP manipulates. More specifically:

1. The definitions of partially-ordered plans and solutions are unchanged.
2. A partial plan is a 4-tuple $\pi = (V, E, act, C)$ as in Equation 3.20, but with two changes. First, in each node v, $act(v)$ may be either a primitive task (an action) or a nonprimitive task. Second, in Equation 3.21, $act(v_2)$ may be either an action, or a goal task that contains $x = b$ or $x \neq b'$.
3. Except for the new definition of a partial plan, partial solutions are the same as in Definition 3.12.

```
Relevant-Methods($\mathcal{M}, \tau$)
    $M \leftarrow \varnothing$
    foreach $m \in \mathcal{M}$ such that task($m$) and task($\tau$) have the same name and
      same number of arguments do
        $m' \leftarrow$ a copy of $m$
        rename the variables in $m'$ to avoid name conflicts with $\tau$
        let $y_1, \ldots, y_n$ be the parameters of task($m'$)
        for $i = 1, \ldots, n$ do
            in $m'$, replace each occurrence of $y_i$ with the $i$'th argument of
              task($\tau$)
        add $m'$ to $M$
```

Algorithm 5.6. Relevant-Methods, which finds a set of methods that are relevant for τ and have no name conflicts. Depending on τ, the methods may be either ground or unground.

4. A partial plan may have three kinds of flaws. In addition to the open goals and threats defined in Section 3.4.2, π has a *compound-task* flaw at every node v such that $act(v)$ is an unrefined compound task. A resolver for this flaw is a relevant method m that has no name conflicts with the variables in τ. Such an m can be found using Relevant-Methods($\mathcal{M}, \tau$), Algorithm 5.6. The flaw can be resolved by replacing v with a sequence of nodes $v_0 \prec \ldots \prec v_k$, where $act(v_0)$ is the goal task pre(m), and $act(v_1), \ldots, act(v_k)$ are the subtasks of m.
5. For open-goal flaws, the "establish p by adding a new action" resolver is not allowed. Instead, p must be established using an action in π. We will loosen this restriction in Section 5.3.

If $P = (\Sigma, s_0, \mathcal{T})$ is a partial-order HTN planning problem, then the following partial solution $\pi_P = (V, E, act, C)$ represents P:

- V includes a node v_0 in which $act(v_0) = a_0$ is a dummy action for s_0 as in Equation 3.20. For each task node $\tau \in \mathcal{T}$, V includes a node v_τ with $act(v_\tau) = \text{task}(\tau)$.
- For each node τ in $\mathcal{T}$ that has no predecessors, E includes an edge (v_0, v_τ). For each precedence constraint $\tau \prec \tau'$ in $\mathcal{T}$, E includes an edge $(v_\tau, v_{\tau'})$.
- $C = \varnothing$.

With the preceding changes, if P is a solvable partial-order HTN planning problem and we call PSP(Σ, π_P), then one or more of its nondeterministic execution traces will return a solution.

5.3 Hybrid HTN/Classical Planning

Because the solutions to an HTN planning problem depend on $\mathcal{M}$, an HTN planner may return failure in some situations where a classical planner would return a solution.

In some cases, this failure may be deliberate. HTN methods can often be a convenient way to encode restrictions that would be more difficult to write as classical action preconditions. For example, if a shuttle bus is supposed to follow a certain route from a to b, then one might write HTN methods that can only produce that route, regardless of whether there are other routes from a to b.

In other cases, the failure might be unintentional. In complicated environments, the methods in $\mathcal{M}$ might not be sufficiently comprehensive to cover every situation that may occur—either because the domain author failed to consider some edge cases, or because the domain author preferred to use a *hybrid planning* approach, that is, to write HTN methods for some of the domain and let the planner use other planning techniques elsewhere.

Hybrid planning is sometimes called *task insertion*, the idea being that the planner can modify the task network by inserting tasks that are not subtasks of anything already in the network. We now give several examples.

```
if $M = \varnothing$ then
    $M \leftarrow Applicable(s) \cup \{m \in Ground(\mathcal{M}) \mid m$ is applicable in $s\}$
```

Algorithm 5.7. Hybrid-planning pseudocode to insert into HTN-Get-Candidates, just after Line 7.

Hybrid forward search. In TO-HTN-Forward and PO-HTN-Forward, inserting a new task into T or $\mathcal{T}$ is roughly equivalent to inserting an action or ground method that is relevant for the task. One possibility is to do this whenever a planning problem would otherwise be unsolvable. For example, the pseudocode in Algorithm 5.7 will modify TO-HTN-Forward and PO-HTN-Forward to add all applicable actions and ground methods to M when it is empty, instead of returning failure. If the modified algorithms are called with $\mathcal{M} = \varnothing$, they will behave like Forward-Search.

```
if $M = \varnothing$ then
    $Landmarks \leftarrow$ RPG-Landmarks$(\Sigma_c, s, g)$
    $M \leftarrow \bigcup_{g' \in Landmarks} Actions(s, g') \cup Methods(s, g', \mathcal{M})$
if $M = \varnothing$ then
    $M \leftarrow Applicable(s) \cup \{m \in Ground(\mathcal{M}) \mid m$ is applicable in $s\}$
```

Algorithm 5.8. Pseudocode for landmark-based hybrid planning, to insert into HTN-Get-Candidates just after Line 7.

We may prefer to add to M only some of the applicable actions and methods, such as the ones that are relevant for some landmarks (defined in Section 3.2.3). If this set is empty, we then can try adding all applicable actions and methods. The pseudocode in Algorithm 5.8 will make TO-HTN-Forward and PO-HTN-Forward do this.

Hybrid plan-space planning. A naive way to get hybrid planning would be to modify the partial-order HTN version of PSP (see Section 5.2.3) to nondeterministically choose, at each iteration of the **while** loop, either to execute lines 1–3 or add to π a nondeterministically chosen task t. However, without any constraints on t, this would work very poorly: there would be an immense state space with a huge branching factor.

Here is a hybrid-planning version of PSP that is much more focused. We take the classical PSP algorithm in Section 3.4 and make all of the changes in Section 5.2.3 with one exception: we omit item 5, that is, we allow the "establish p by adding a new action" resolver to be used. This gives the algorithm all of the plan-space planning capabilities in Section 3.4. We also add the following resolver for open goals, so that the algorithm can use goal methods to resolve them:

- *Establish p using a goal method.* Let p be an open goal, and m be a standardized[6] goal method such that p is an instance of $\text{task}(m)$. Then the following modification to π is a resolver for p:

  Add to π a new node v' with $act(v') = m$, instantiate variables of m to make $\text{task}(m)$ match p, add a causal link $v' \overset{p}{\dashrightarrow} v$, and add edges (v_0, v') and (v', v) to E so that $v_0 \prec v' \prec v$.

5.4 Heuristics, Expressivity, Complexity

Here are brief discussions of heuristic functions, expressivity, and computational complexity for HTN planning.

Heuristic functions. Section 3.1.4 described how to use heuristic functions for node selection in classical planning algorithms, and heuristic functions similarly can be used for node selection in TO-HTN-Forward-Det and PO-HTN-Forward. It would not work well to take the heuristic functions in Section 3.2 and use them directly, but there are ways to translate them into heuristic functions for HTN planning. The details of those translations are rather complicated.

Heuristic functions can also be developed using a data structure called a *task decomposition graph*, an And/Or graph that is like a union of all the possible refinement trees for the planning problem. There is a root node similar to the one in a refinement tree. For each ground task t, there is an Or-node whose children are all of the applicable ground methods relevant for t. For each ground method, there is an And-node whose children are the subtasks. The graph can be created once when the planning domain is defined, and searched whenever heuristic values are needed. We omit the details of these approaches, but Section 5.6.3 cites relevant publications.

Expressivity. partial-order HTN planning is more expressive than total-order HTN planning, which is more expressive than classical planning. The details of the proof

[6]That is, the object variables in m have been renamed to avoid name conflicts.

depend on the theory of formal languages, but the basic idea is as follows. partial-order HTN planning has equivalent expressive power to context-sensitive languages: each partial-order HTN planning problem's set of solutions corresponds to a context-sensitive language, and vice versa. Similarly, total-order HTN planning has equivalent expressive power to context-free languages, and classical planning has equivalent expressive power to regular languages. There are context-sensitive languages that are not context-free, and context-free languages that are not regular languages.

Several subsets of total-order HTN planning can be translated to classical planning. Total-order HTN planning can also be translated into propositional logic. These translation techniques have been used as a basis for efficient total-order HTN planners (see Section 5.6.1).

Computational complexity. Undecidable problems can be encoded as total-order HTN planning problems, and thus also as partial-order HTN planning problems. Again we will skip the details of the proof, but it involves taking a well-known undecidable problem—whether two context-free languages have a nonempty intersection—and encoding it as a total-order HTN planning problem.

Hybrid total-order HTN/classical planning is EXPSPACE-complete, and hybrid partial-order HTN/classical planning is 2-NEXPTIME-complete. These complexity results are intermediate between the complexity of classical planning and HTN planning.

As in Section 2.5, these are *worst-case* results. In many planning domains the time complexity is much lower: many are polynomial in the average case, and some are polynomial even in the worst case.

5.5 Refinement of Abstract Actions

Another variant of HTN planning is to refine *abstract actions*. Like tasks, these are complex activities that need to be accomplished, but they have preconditions and effects somewhat like those of the non-abstract actions in Section 2.3.2.

5.5.1 Representation

We will represent abstract actions as ground instances of *abstract-action (AA) schemas*, where each such schema is a triple $\alpha = (\text{head}(\alpha), \text{pre}(\alpha), \text{eff}(\alpha))$. Although AA schemas and abstract actions may appear syntactically identical to their non-abstract counterparts, their semantics is different. In an abstract action $a = (\text{head}(a), \text{pre}(a), \text{eff}(a))$,

- $\text{head}(a)$ is a task to be refined,
- $\text{pre}(a)$ is a precondition that must be true when the task begins,
- the "effect" $\text{eff}(a)$ is similar to a goal: it is a condition that must be true when the task finishes.

Given a state s that satisfies $\text{pre}(a)$, a planner will try to refine a into a plan π such that $\gamma(s, \pi) \models \text{eff}(a)$.

For simplicity of presentation, we will focus on sets of abstract actions that are totally ordered. A *total-order AA method* is a tuple

$$m = (\mathrm{head}(m), \mathrm{task}(m), \mathrm{pre}(m), \mathrm{subtasks}(m), \mathrm{eff}(m)), \tag{5.14}$$

such that

- $\mathrm{head}(m)$, $\mathrm{task}(m)$, and $\mathrm{pre}(m)$ are as in Definition 5.2.
- $\mathrm{subtasks}(m)$ is a sequence of actions. Each action may be lifted, ground, or partially instantiated, and may be either abstract or non-abstract.
- $\mathrm{eff}(m)$ is a condition that must be true after completion of $\mathrm{subtasks}(m)$.

A ground method m is *relevant* for an abstract action a if $\mathrm{task}(m) = \mathrm{head}(a)$ and $\mathrm{eff}(m) \models \mathrm{eff}(a)$. By analogy to Equation 5.4, if g is a set of literals, a is an abstract action and $\mathcal{M}$ is a set of total-order AA methods, then

$$\mathit{Methods}(g, a, \mathcal{M}) = \{m \in \mathit{Ground}(\mathcal{M}) \mid g \models \mathrm{pre}(m) \text{ and } m \text{ is relevant for } a\}. \tag{5.15}$$

A *total-order AA planning domain* is a triple $\Sigma = (\Sigma_c, \mathcal{A}, \mathcal{M})$, where Σ_c is a classical planning domain, $\mathcal{A}$ is a set of abstract actions, and $\mathcal{M}$ is a set of AA methods. An *abstract plan* is a sequence of actions, each of which may be either abstract or non-abstract. A *total-order AA planning problem* is a tuple $P = (\Sigma, s_0, A)$, where Σ is a total-order AA planning domain, s_0 is an initial state, and A is an abstract plan.

A *solution* for a total-order AA planning problem $P = (\Sigma, s_0, A)$ is a non-abstract plan that is defined inductively as follows. If $A = \langle\rangle$, then A itself is a solution for P. Otherwise, let a be the first action in A, so that $A = a \cdot A'$ for some A'. Then:

1. If a is non-abstract and is applicable in s_0, then for every solution π for $(\Sigma, \gamma(s_0, a), A')$, the plan $a \cdot \pi$ is a solution for P.
2. If a is abstract and $m \in \mathit{Methods}(s, a, \mathcal{M})$, then every solution for $(\Sigma, s_0, \mathrm{subtasks}(m) \cdot A')$ is also a solution for P.

If A is a non-abstract plan, it follows from the definition that if A is applicable in s_0 then A is P's only solution, and otherwise P has no solution.

5.5.2 Adaptations of HTN Algorithms

It is straightforward to write total-order AA adaptations of the planning algorithms in Sections 5.1, 5.2, and 5.3 and the heuristic functions in Section 5.4. In Algorithm 5.9, which is based on TO-HTN-Forward, $\Sigma = (\Sigma_c, \mathcal{A}, \mathcal{M})$ is the planning domain and $P = (\Sigma, s_0, A)$ is the planning problem. If P is solvable, then at least one of the nondeterministic traces will return a solution. Otherwise they all will return failure.

By analogy to Section 5.1.4, *serial solvability* of P is defined recursively as follows. If $A = \langle\rangle$ then P is serially solvable. Otherwise, let a be the first element of A, so that $A = a \cdot A'$ for some A'. Then P is serially solvable if for every plan π that solves $(\Sigma, s_0, \langle a\rangle)$, the planning problem $(\Sigma, \gamma(s_0, \pi), A')$ is serially solvable. TO-HTN-Serial can easily be modified to produce a TO-AA-Serial algorithm for serially-solvable total-order AA planning problems.

```
TO-AA-Forward($\Sigma_c, \mathcal{A}, \mathcal{M}, s, A$)
    if $A$ is empty then return $\langle\rangle$
    $a \leftarrow$ the first element of $A$;  $A' \leftarrow$ the rest of $A$
    if $a$ is abstract then
1       $M \leftarrow Methods(s, a, \mathcal{M})$
        if $M \neq \varnothing$ then
2           nondeterministically choose $m \in M$
3           return TO-AA-Forward($\Sigma_c, \mathcal{A}, \mathcal{M}, s$, subtasks$(m) \cdot A'$)
    else if $a$ is applicable in $s$ then
        $\pi \leftarrow$ TO-AA-Forward($\Sigma_c, \mathcal{A}, \mathcal{M}, \gamma(s, a), A'$)
        if $\pi \neq$ failure then return $a \cdot \pi$
    return failure
```

Algorithm 5.9. TO-AA-Forward, an AA adaptation of TO-HTN-Forward.

5.5.3 Angelic Refinement

A total-order AA domain $\Sigma = (\Sigma_c, \mathcal{M})$, is *downward refinable* if for every abstract action a in Σ and every state s that satisfies pre(a), $Methods(s, a, \mathcal{M})$ is nonempty. In other words, whenever $s \models$ pre(a) there is at least one $m \in Ground(\mathcal{M})$ such that

$$\text{task}(m) = \text{head}(a), \quad s \models \text{pre}(m), \quad \text{and} \quad \text{eff}(m) \models \text{eff}(a).$$

In a downward-refinable total-order AA domain, an abstract plan A is *angelically refinable* if A is empty, if A contains a single action, or if A is a sequence of n actions $A = \langle a_1, a_2, ..., a_n\rangle$ such that

$$\text{eff}(a_1) \models \text{pre}(a_2), \quad \text{eff}(a_2) \models \text{pre}(a_3), \quad \ldots, \quad \text{eff}(a_{n-1}) \models \text{pre}(a_n).$$

If A is angelically refinable then we define

$$\text{pre}(A) = \begin{cases} \text{pre}(a), & \text{if } a \text{ is the first action in } A, \\ \varnothing, & \text{if } A = \langle\rangle. \end{cases}$$

A planning problem $P = (\Sigma, s_0, A)$ is *angelically solvable* if A is angelically refinable and $s_0 \models$ pre(A). Obviously, most total-order AA planning problems are not angelically solvable—but the ones that are can be solved very quickly.

If P is angelically solvable then it is serially solvable, so it can be solved by the TO-AA-Serial algorithm mentioned at the end of Section 5.5.2. However, it also can be solved more simply using Algorithm 5.10, TO-AA-Angelic.

TO-AA-Angelic is similar to TO-HTN-Serial and TO-AA-Serial, but it omits several of the failure tests and makes a deterministic rather than nondeterministic choice. For angelically solvable planning problems, it will always return solutions, and for unsolvable problems it will always return failure. For planning problems that are solvable but not angelically solvable, it might or might not return solutions.

```
TO-AA-Angelic($\Sigma_c, \mathcal{M}, s, A$)
    $\pi \leftarrow \langle\rangle$
    while $A \neq \langle\rangle$ do
1       $a \leftarrow$ the first action in $A$;  $A \leftarrow$ the rest of $A$
        if $a$ is abstract then
2           arbitrarily select any $m \in Methods(s, a, \mathcal{M})$
            $\pi' \leftarrow$ TO-AA-Angelic($\Sigma_c, \mathcal{M}, s$, subtasks$(m)$)
            $s \leftarrow \gamma(s, \pi')$;  $\pi \leftarrow \pi \cdot \pi'$
        else if $a$ is applicable in $s$ then
            $s \leftarrow \gamma(s, a)$;  $\pi \leftarrow \pi \cdot a$
        else return failure
    return $\pi$
```

Algorithm 5.10. TO-AA-Angelic, a planner for angelically refinable domains.

We have only defined angelic refinability and solvability for total-order AA planning domains. However, it is possible to write similar definitions for total-order HTN planning domains, and a TO-HTN-Angelic algorithm similar to TO-AA-Angelic.

5.6 Discussion and Bibliographic Notes

5.6.1 General Background

The first HTN planning systems began to be developed in the mid-1970s [970, 1079]. They did plan-space HTN planning in a manner somewhat like the algorithm in Section 5.2.3, but with several other elaborate features [1172, 264]. There also was work that used planning-graph techniques to make plan-space HTN planning more efficient [737, 736].

In addition to preconditions, some plan-space HTN planners allowed methods to have *filter* conditions [327]. These look syntactically like preconditions—but instead of treating them as goals to achieve, the planner would not use a method if its filter conditions were not true. However, determining the correct state in which to evaluate filter conditions is a complicated issue [327], and most current HTN planners do not use them (an exception is [682]).

In an effort to simplify HTN planning, several HTN planners used forward-search instead of plan-space planning, and removed various of the complicating features of the early HTN planners, including filter conditions and goal tasks [832, 833, 410]. This simplified model of HTN planning became highly influential. Later research efforts, having lost track of goal tasks as a part of HTN planning, re-invented them under the name of *goal-network planning* [1009, 1010]. Because our HTN formulation includes goal tasks (see Section 5.1), it has a version of goal-network planning as a special case. TO-HTN-Forward and PO-HTN-Forward are based loosely on the SHOP [832], SHOP2 [833], and GDP [1009] algorithms.

Several other HTN planning techniques involve translating HTN planning problems into some other format. For some restricted classes of HTN planning, planning problems can be translated into classical planning problems and solved using classical planners [24, 512]. Another approach is to translate HTN planning problems into satisfiability problems, and construct plans using a satisfiability solver [107, 108, 105, 989]. This uses a version of iterative deepening: it bounds the length of the solution and uses the satisfiability solver to see whether there's a solution of that length—and if not, then it increases the bound and tries again. HTN instances can also be compiled into programs in a conventional programming language [534, 747]. This can, for example, allow optimizations to minimize backtracking [747].

An HTN planner performed quite well in the 2002 International Planning Competition [734], but HTN planners were not involved in subsequent competitions until 2020. The 2020 International Planning Competition focused entirely on HTN planning, and stimulated research on several new HTN planners, including several of the ones cited in the preceding paragraphs. Our formulation of partial-order HTN planning is based loosely on the HDDL language that was developed for the competition [515].

5.6.2 Formal Models

Theoretical models for HTN planning began to be developed in the early 1990s [1198, 577]. Subsequent theoretical investigations have produced formal semantics for plan-space [327] and forward-search [430] HTN planning, a provably correct planning algorithm [328], and analyses of the complexity and expressivity of HTN planning [330, 513] and hybrid HTN/classical planning [27].

Our presentation of HTN planning algorithms has focused primarily on forward search and plan-space search, but those are not the only possibilities. A formal model of HTN search spaces [26] has shown that because they have a more complex structure than classical search spaces, there is a wider variety of possible ways to search them, including some possibilities for which no planning algorithms have yet been written.

5.6.3 Heuristic Functions

The complex refinement structure of HTN planning domains makes it more difficult to develop heuristic functions than in classical planning. One approach is to translate the HTN planning problem into a classical planning problem for which classical planning heuristics can be used: [28] does this for a subset of HTN planning problems in which the method calls are tail-recursive, and [514] does it by encoding a relaxed version of the HTN planning problem as a classical planning problem. There also are ways to adapt classical heuristics for use in HTN planning [514].

In [114], task decomposition graphs are used as the basis for an admissible HTN heuristic. Task decomposition graphs have also been used to develop heuristic functions for hybrid HTN/classical planning [113].

5.6.4 Hybrid HTN/Classical Planning

Some of the early work on HTN planning used a hybrid approach that combined HTN planning with classical plan-space planning [328, 581], and this approach was also used in several subsequent works [737, 736, 144, 113, 324]. Hybrid HTN/classical planning is roughly equivalent to *HTN planning with task insertion*, in which arbitrary tasks may be inserted into the task network [396]. Inserting a task into a task network is equivalent to inserting the task's methods and actions as we did in Algorithm 5.7.

Section 5.3 mentioned the possibility of using a landmark computation such as RPG-Landmarks to identify goals that match PO-HTN-Forward's goal methods. The GoDeL planner [1010] used this approach. If GoDeL encountered a goal for which there is an applicable method then it would use the method, and otherwise it would invoke a landmark-based forward search. During each episode of landmark generation, GoDeL treated the landmarks as intermediate goals, and reverted to HTN planning whenever it encountered a landmark for which there was an applicable method.

Another approach to hybrid planning is to run a classical planner and an HTN planner as two separate subroutines, with the HTN planner passing control to the classical planner whenever it encounters a task for which no methods have been defined, and the classical planner passing control to the HTN planner whenever it encounters an "action" that matches an HTN method [402]. A similar effect has been achieved by compiling a set of HTN methods (subject to certain restrictions) into a set of classical "actions" whose names, preconditions, and effects encode the steps involved in applying the methods, and using these actions in a classical planner [24].

5.6.5 Planning with Abstract Actions

In Section 5.5.3, downward refinability of a planning domain is an HTN version of the downward refinement property in Section 3.6.8. Angelic refinability of a plan is related to *angelic nondeterminism* in the theory of programming [115, 148], where an angelically nondeterministic "choose" command is assumed to make a successful choice if one exists.[7] In top-down program design, one can first write a high-level algorithm that contains a "choose," and then replace the "choose" with deterministic code for making the best choice. That is how we wrote the TO-AA-Angelic algorithm.

TO-AA-Angelic was inspired by the Angelic-Search algorithm in [967, Section 11.4], but there is a key difference: unlike TO-AA-Angelic, which always refines the first action in A before refining the rest of A, Angelic-Search may refine the actions of A in any order. To illustrate a difficulty that this causes, let us suppose that $A = a_1 \cdot a_2$, where both a_1 and a_2 are abstract, and suppose we want to refine a_2 before a_1. If M is the set of relevant methods for a_2, then it is unclear which $m \in M$ to choose. If we make the wrong choice, then the refined problem $P' = (\Sigma, s_0, a_1 \cdot \text{subtasks}(m))$ might not be solvable. The discussion of Angelic-Search includes some clever ways to deal with this problem by matching sets of refinements of a_1 with sets of refinements of a_2. However, the details are complicated and are not always feasible to implement.

[7] The **nondeterministically choose** command in our pseudocode algorithms is very similar; see Section A.1 for details.

5.6.6 Extensions and Applications

In HTN planners based on forward search, it is easy to allow arbitrary computational formulas in the methods and action schemas. Several total-order HTN planners [832, 1009, 830], and a few partial-order HTN planners [833, 433], work this way. This makes it possible to use application-specific data structures. Extensions have also been developed for Horn-clause inference [833], temporally-extended preferences [1043], temporal planning (TemPlan in Chapter 17), probabilistic environments (UPOM in Chapter 15), and coordination of multi-agent systems [697].

Such extensions, along with the ability of HTN methods to represent "standard operating procedures" [1170], have helped to make HTN planners useful in a variety of applications. Some examples include scheduling [1171], logistics and crisis management [265, 1080, 144], spacecraft planning and scheduling [1, 332], equipment configuration [13], manufacturing process planning [1039], evacuation planning [821], real-time tracking [520], composition of web services [1031, 1042] and information streams [1044], requirements engineering [1043], computer games [1040, 504, 214, 784, 840], and robotics [809, 566, 1059, 277, 143].

One difficulty in developing HTN-planning applications is that the domain author needs to write and debug a potentially complex set of domain-specific HTN methods [575]. A potential way to alleviate this problem is to learn HTN methods automatically. Chapter 7 describes some of the research on that topic.

5.6.7 Other Topics

Plan verification. HTN plan verification consists of two closely-related problems:

- Given an HTN planning problem $P = (\Sigma, s_0, T)$ and a plan π, is π a solution for P?
- Given an HTN planning domain Σ and a plan π, does there exist a planning problem $P = (\Sigma, s_0, T)$ such that π is a solution for P?

If Σ were a classical planning domain, then both problems could be solved in low-order polynomial time by evaluating $\gamma(s_0, \pi)$. However, verification of HTN planning problems is much more complicated because of the need to check whether π can be produced by HTN refinement.

One approach to HTN plan verification is to translate the planning problem into a Boolean formula that is satisfiable if and only if π is a solution for P [106]. Another is to translate Σ into an attribute grammar and check whether π can be parsed as a solution for P [94, 95].[8] A third approach is to translate the verification problem into an HTN planning problem that can be solved by an HTN planner [517].

5.7 Exercises

5.1. In Example 5.4, rewrite m2-get-container to eliminate the parameter p by replacing it with pile(c).

[8] An extended version of this approach can try to correct a plan by deleting actions from it [96].

5.2. Modify the following methods to satisfy the restrictions given in Remark 2.6: m2-uncover in Example 5.4, m3-navigate in Example 5.4, and m1-put-on-robot in Example 5.10.

5.3. Write methods for some situations that m2-get-container in Example 5.4 doesn't handle, such as the case where cargo(r) is neither nil nor c.

5.4. Characterize the situations that the uncover methods in Example 5.4 don't handle. Write methods to handle those cases.

5.5. Prove that TO-HTN-Forward is sound and complete. Do the same for PO-HTN-Forward.

5.6. Write total-order HTN methods for the usual recursive formulation of the Towers of Hanoi problem. Using your methods, is the problem serially solvable? Why or why not?

5.7. In Example 5.10, write an m2-put-on-robot method for the case where c is not at the top of its pile.

5.8. In Example 5.10, write an m-remove-from-robot method in which a crane removes a container from a robot and puts it onto a designated pile. If this is available, then does it change the number of solution plans in Example 5.12? Why or why not?

5.9. In Example 5.10, write methods for put-on-robot(c, r) for these cases: (a) k is empty but r is not; (b) r is empty but k is not; (c) neither r nor k is empty.

5.10. Write a partial-order HTN planning algorithm similar to TO-HTN-Forward-RT.

5.11. Professor Prune claims that any partial-order HTN method can be translated into an equivalent set of methods with totally ordered subtasks. Here is his argument:

> Let $m = (\text{head}(m), \text{task}(m), \text{pre}(m), \text{subtasks}(m), \prec_m)$ be the partial-order HTN method, and let $\prec_1, \ldots, \prec_k$ be all of the total orderings that satisfy $\prec_m$. For $i = 1, \ldots, k$, let $m_i = (\text{head}(m), \text{task}(m), \text{pre}(m), \prec_i)$. Then $m_1, \ldots, m_k$ can produce every totally ordered solution plan that m can produce.

To show that Professor Prune is wrong, write a partial-order HTN method m that can produce a solution plan that none of $m_1, \ldots, m_k$ can produce.[9]

5.12. Definition 5.11 defined partial-order HTN solution plans to be totally ordered. Write a definition of a partially ordered solution plan, and modify PO-HTN-Forward to find such solutions.

5.13. In Section 3.4.2, there were two open-goal resolvers: one that uses an action already in π, and one that adds a new action to π.

(a) Why does Section 5.2.3 use only one of those resolvers?

[9]Thanks to Pascal Bercher for inspiring this exercise.

(b) In Section 5.3 there is an open-goal resolver that uses a goal method. Why aren't there two such resolvers, like there are for actions?

5.14. Prove that if If $P = ((\Sigma_c, \mathcal{M}), s_0, A)$ is a solvable total-order AA planning problem, then at least one of the nondeterministic traces of TO-AA-Forward$(\Sigma_c, \mathcal{M}, s_0, A)$ will return a solution plan, and otherwise all of the traces will return failure.

5.15. Write total-order AA adaptations of one or more of the following algorithms: TO-HTN-Forward-Det, TO-HTN-Forward-RT, TO-HTN-Serial, PO-HTN-Forward, the hybrid planning algorithms in Section 5.3, and the heuristic functions in Section 5.4.

5.16. Prove that if P is an angelically solvable planning problem, it is serially solvable.

5.17. Prove that if P is an angelically solvable planning problem, TO-AA-Angelic will find a solution for P.

5.18. Analyze the time complexity of TO-AA-Angelic.

5.19. In a downward-refinable planning domain, consider solvable planning problems that are not angelically solvable.

(a) Give an example of such a problem.

(b) Will TO-AA-Angelic always (or ever) solve such problems? Why or why not?

5.20. Write definitions of downward refinability, angelic refinability, and angelic solvability for total-order HTN planning domains. Write a TO-HTN-Angelic algorithm similar to TO-AA-Angelic.

6 Acting with HTNs

This chapter discusses how to use HTN domain models during acting. One of the biggest issues, of course, is that unlike an HTN domain model, the actor's environment is not necessarily deterministic or static: exogenous events and unanticipated action outcomes can make the current state different from what an HTN model would predict. Despite this, an HTN domain model can be a very useful way to provide some of the actor's *know-how*—the operational information discussed in Section 1.2. HTN methods can provide instructions to the actor on how to perform complex tasks without the overhead of searching through a large state space, how to avoid situations where unanticipated events are likely to cause bad outcomes, and how to recover when unanticipated events occur.

Section 6.1 describes a way to use HTN methods for purely reactive acting, and some potential problems with this approach. Section 6.1 describes some simple ways for an actor to use an HTN planner, replanning if problems occur. Section 6.2.1 describes ways to repair existing plans when unexpected events occur during acting. Finally, Section 6.3 is the discussion and bibliographic remarks and Section 6.4 is the exercises.

```
TO-HTN-Act($\Sigma_c, \mathcal{M}, T$)
     if $T$ is empty then return success
     $t \leftarrow$ the first element of $T$;  $T' \leftarrow$ the rest of $T$
1    $s \leftarrow$ observe current state
     $M \leftarrow$ HTN-Get-Candidates($\Sigma_c, \mathcal{M}, s, t$)
2    foreach $m \in M$ do
         if $m$ is a method then
             if TO-HTN-Act($\Sigma_c, \mathcal{M}$, subtasks($m$)$\cdot T'$) = success then return
               success
         else if $m$ is an action then
3            execute $m$
             if $m$ executed successfully then return TO-HTN-Act($\Sigma_c, \mathcal{M}, T'$)
     return failure
```

Algorithm 6.1. TO-HTN-Act, a reactive HTN acting algorithm. The HTN-Get-Candidates subroutine is the same as in Algorithm 5.1.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

6.1 Reactive HTN Acting

Algorithm 6.1, TO-HTN-Act, is a modified version of TO-HTN-Forward that uses HTN methods for acting instead of planning. The modifications are as follows:

- Instead of taking s as an argument, TO-HTN-Act observes s in Line 1. Instead of computing γ in Line 3, TO-HTN-Act performs the action on its execution platform. Instead of returning a plan, it returns either success or failure.
- The loop at Line 2 is a failure-recovery mechanism. If failure occurs when using a chosen method m, TO-HTN-Act tries to reaccomplish T using other methods in M. If all of the other methods fail, then it returns failure to the next higher level in the recursion stack, which will try to reaccomplish the task that had t as a subtask. The approach is similar to backtracking, but not identical. At each loop iteration after the first one, a true backtracking algorithm would revert s to the value it had before the first loop iteration, but TO-HTN-Act cannot time-travel back to a state in the past.

Similar modifications can be used to transform other planning algorithms in Chapter 5 into acting algorithms.

As written, TO-HTN-Act operates purely reactively. Instead of looping through the methods in an arbitrary order in Line 2, it may perform better if it can make an informed choice of which $m \in M$ to try first. One possibility is to use a heuristic function. Another is for it to choose m by calling an online HTN planner that returns the topmost method in its refinement tree (most of the planners in Chapter 5 can easily be modified to do this). However, this can cause repeated duplication of effort between TO-HTN-Act and the planner, increasing their time complexity.[1] The next section discusses some acting algorithms that operate more efficiently.

6.2 Acting with an Online HTN Planner

Algorithms 6.2 and 6.3, HTN-Run-Lookahead and HTN-Run-Lazy-Lookahead, are receding-horizon actors that get their plans from an HTN planner. They are modified versions of Run-Lookahead and Run-Lazy-Lookahead in Section 2.6.2. The modifications are as follows:

- Σ and $\mathcal{T}$ may be either a total-order HTN planning domain and sequence of tasks as in Section 5.1, or a partial-order HTN planning domain and task network as in Section 5.2. In either case, the planning problem is $(\Sigma, s, \mathcal{T})$, where s is the current state.
- *Lookahead* is an online HTN planner. This could be one of the planning algorithms in Chapter 5, possibly modified to terminate its search early. For

[1] Suppose TO-HTN-Act chooses m by calling an HTN planner M that returns a method, and let *Tree* be the refinement tree for m. Then TO-HTN-Act's recursive calls will be like a preorder traversal of *Tree*. At each nonprimitive task t, M will be called again and will redo its search of the subtree below t. In the worst case, these unnecessary calls to M can increase the time complexity by a multiplicative factor of n, where n is the number of nonprimitive tasks in *Tree*.

example, in a receding-horizon approach, *Lookahead* could search to a cutoff depth and return the best partial plan that it has seen so far.

- Since HTN planning does not necessarily have a goal state, the termination criterion is whether *Lookahead* returns $\langle\rangle$. When implementing Lookahead, one should ensure that it returns $\langle\rangle$ only when nothing needs to be done.
- For simplicity of presentation, HTN-Run-Lazy-Lookahead does not call a *Simulate* subroutine. However, it can easily be modified to do so.

```
HTN-Run-Lookahead($\Sigma, \mathcal{T}$)
    while True do
1       $s \leftarrow$ observed current state
2       $\pi \leftarrow$ Lookahead($\Sigma, s, \mathcal{T}$)
        if $\pi$ = failure then return failure
3       if $\pi = \langle\rangle$ then return success
        $a \leftarrow$ pop($\pi$)      // remove and return $\pi$'s first action
4       trigger execution of $a$
```

Algorithm 6.2. HTN-Run-Lookahead, which replans at each action.

```
HTN-Run-Lazy-Lookahead($\Sigma, \mathcal{T}$)
    $\pi \leftarrow \langle\rangle$
    while True do
        if $\pi = \langle\rangle$ then
1           $s \leftarrow$ observed state
2           $\pi \leftarrow$ Lookahead($\Sigma, s, \mathcal{T}$)
            if $\pi$ = failure then return failure
3           if $\pi = \langle\rangle$ then return success
        $a \leftarrow$ pop($\pi$)      // remove and return $\pi$'s first action
        trigger execution of $a$
```

Algorithm 6.3. HTN-Run-Lazy-Lookahead, which replans only when necessary.

HTN-Run-Lookahead and HTN-Run-Lazy-Lookahead can work well in some cases and badly in others. Here are examples of both cases:

Example 6.1. In Example 5.7, consider $P = (\Sigma, s_0, \langle \text{put-in-pile}(\text{c1}, \text{p}_2)\rangle)$ as an acting problem, with *Lookahead* = TO-HTN-Forward. Let us consider two cases: (i) if the state-transition function's predictions are completely accurate, so that the current state after executing actions $a_1, \ldots, a_n$ is always $\gamma(s_0, \langle a_1, \ldots, a_n\rangle)$; and ($ii$) if there are some failures or unexpected events but they do not make the problem unsolvable.

In case (i), here is what will happen if we use HTN-Run-Lazy-Lookahead:

- In the first loop iteration, $\pi = \langle\rangle$ and $s = s_0$. HTN-Run-Lazy-Lookahead calls

TO-HTN-Forward(Σ, s_0, ⟨put-in-pile(c1, p_2)⟩), which returns

$$\pi = \langle \text{take(r1, c1, c2, p1, d1), move(r1, d1, d2), put(r1, c1, c3, p2, d2)} \rangle.$$

From π, HTN-Run-Lazy-Lookahead pops and executes take(r1, c1, c2, p1, d1).

- In the 2nd and 3rd loop iterations, HTN-Run-Lazy-Lookahead pops and executes move(r1, d1, d2) and put(r1, c1, c3, p2, d2). This leaves $\pi = \langle\rangle$.
- In the 4th loop iteration, HTN-Run-Lazy-Lookahead calls TO-HTN-Forward, which returns $\pi = \langle\rangle$. HTN-Run-Lazy-Lookahead exits with success.

If we instead use HTN-Run-Lookahead, it also will execute the same actions and return success, but it will call TO-HTN-Forward once before executing each action.

In case (*ii*), the task-list methods in Σ (see Example 5.4) are robust enough that in most cases, both HTN-Run-Lazy-Lookahead and HTN-Run-Lookahead will recover and finish successfully. □

Example 6.2. HTN-Run-Lookahead and HTN-Run-Lazy-Lookahead have more difficulty with the planning problem in Example 5.12. As before, suppose *Lookahead* is PO-HTN-Forward and it returns the following solution from Example 5.12,

$$\pi_2 = \text{unstack(k2,c1,c2,p2,d2), move(r1,d1,d2), load(k2,c1,r1,d2)}.$$

If no unexpected events occur during execution, then Run-Lazy-Lookahead will execute π_2 to completion and return success, but Run-Lookahead will not. Run-Lookahead will execute the unstack action, then call PO-HTN-Forward again, which will fail because no methods are applicable when k2 is holding c1.

If execution failures occur that are not serious enough to make the problem unsolvable, HTN-Run-Lazy-Lookahead will succeed in some cases and fail in others. As an example of the latter, suppose a transient error causes move(r1,d1,d2) to fail without changing the current state. Then HTN-Run-Lazy-Lookahead will call PO-HTN-Forward again, which will fail because no methods are applicable when k2 is holding c1. □

6.2.1 Acting with Plan Repair

When an actor executes a plan, execution errors or exogenous events may sometimes cause it to fail. In Section 6.2, HTN-Run-Lookahead and HTN-Run-Lazy-Lookahead recover from such failures by discarding the old plan and calling the planner again, giving it the same task as before and the current observed state. However, as we discussed in Section 3.5, there are several reasons why it may be preferable to repair the original plan instead.

The plan-repair algorithm in Section 3.5 used a relatively simple heuristic for deciding what parts of the plan to try to repair. In HTN planning domains, the planning problem's refinement structure can help decide what repairs to try. Algorithm 6.4, HTN-Run-Repair, is a modified version of HTN-Run-Lazy-Lookahead that does this.

HTN-Run-Repair begins by calling *Lookahead-RT*, which may be any online planner that returns a refinement tree, such as an online version of TO-HTN-Forward-RT.

```
HTN-Run-Repair($\Sigma, T$)
    $s \leftarrow$ observed state
    $tree \leftarrow$ Lookahead-RT($\Sigma, s, T$)
    if $tree$ = failure then return failure
    $\pi \leftarrow$ the sequence of action nodes in $tree$
    while True do
        if $\pi = \langle\rangle$ and $tree$ has no unsolved nodes then return success
        $v \leftarrow$ pop($\pi$)      // the first action node in $\pi$
        trigger execution of $content(v)$
        if $content(v)$ failed then
            // repair the tree, and make $v'$ the next action node to perform
            $v' \leftarrow$ HTN-Repair($\Sigma, v_0, v$)
            if $v'$ = failure then return failure
            $\pi \leftarrow$ the sequence of action nodes starting at $v'$
```

Algorithm 6.4. HTN-Run-Repair, which repairs its plan if actions fail.

After *Lookahead-RT* returns, HTN-Run-Repair starts executing the actions in the refinement tree's action nodes. If it encounters a failure or the refinement tree ends, it calls *HTN-Repair*, which may be any algorithm that repairs the part of the refinement tree where the failure occurred and returns the node at which execution should resume (Section 6.2.2 will discuss such algorithms). This process repeats until either HTN-Run-Repair finishes successfully or a failure occurs that *HTN-Repair* cannot fix.

Example 6.3. Suppose we run HTN-Run-Repair on the planning problem in Example 5.12, and *Lookahead-RT* returns the root of the refinement tree for π_2 in Figure 5.4. When HTN-Run-Repair tries to execute π_2, suppose unstack(k2,c1,c2,p2,d2) finishes correctly but move(r1,d1,d2) fails. Then HTN-Run-Repair will call *HTN-Repair*(Σ, v_0, v), where v_0 is the root node and $v = (a_{22}, \text{move(r1,d1,d2)})$ is the node where the failure occurred. Suppose *HTN-Repair* decides that the failure was transient and the action should be tried again. Then *HTN-Repair* will return v and HTN-Run-Repair will try to execute move(r1,d1,d2) again. If there are no further problems, it will execute the rest of the refinement tree to completion. □

6.2.2 Incremental Plan Repair

This section discusses some repair algorithms that can be used as HTN-Run-Repair's *HTN-Repair* algorithm.

Let *Tree* be a refinement tree, and v be a task node where a failure has occurred during acting. Algorithm 6.5, Incremental-Repair, looks for a new plan for v that will preserve applicability of the rest of *Tree*. If it cannot find one, then it calls itself recursively on the next higher task node, to replan a larger part of *Tree*.

Incremental-Repair-2, Algorithm 6.6, is a finer-grained version of Incremental-Repair. It first looks for a plan for v that will preserve applicability of the rest of *Tree*.

```
HTN-Incremental-Repair($\Sigma$, $Tree$, $v$)
    $\pi_{\text{pending}} \leftarrow$ the sequence of actions after $v$ in $Tree$
    $s \leftarrow$ observed current state
    $t \leftarrow content(v)$
    $Tree_1 \leftarrow$ Lookahead-RT($\Sigma, s, \langle t \rangle$)          // a new refinement tree for t
    if $Tree_1 \neq$ failure then
        $\pi_1 \leftarrow$ the sequence of actions in $Tree_1$'s leaf nodes
        if $\pi_{\text{pending}}$ is applicable in $\gamma(s, \pi_1)$ then
            $v_1 \leftarrow$ the task node for $t$ in $Tree_1$
            $v_{\text{first}} \leftarrow$ the first action node in $Tree_1$
            in $Tree$, replace $v$ with $v_1$
            return $v_{\text{first}}$                                  // the action to execute next
    else
        if $parent(v)$ is the root of $Tree$ then return failure
        $v' \leftarrow$ the lowest ancestor of $v$ that is a task node
        return Incremental-Repair($\Sigma$, $Tree$, $v'$)
```

Algorithm 6.5. HTN-Incremental-Repair, which replans increasingly larger tasks. At the failed task node v, it looks for a new plan π_1 that preserves applicability of the actions after v. If that fails, it calls itself recursively on the task node above v.

If that fails, it tries to find a plan for both v and the next sibling node (if there is one) that will preserve applicability of the rest of *Tree*. If that fails, it tries planning for v and its next two siblings, and so forth until there are no more siblings. If that fails, it calls itself recursively on the next higher task node, to replan a larger part of *Tree*.

If one of the preceding algorithms is HTN-Run-Repair's *HTN-Repair* subroutine, it can sometimes happen that HTN-Run-Repair repeatedly encounters a failure at some node v, and *HTN-Repair* repeatedly returns the same repair that failed the previous time. Sometimes this may be the right thing to do: if the repaired plan's failure is transient then one might want to keep trying until it succeeds. However, if the repaired plan's failure is inevitable then *HTN-Repair* should try a different repair. For the latter case, Incremental-Repair and Incremental-Repair-2 can be modified to keep a list of the repairs that they have already tried, and not try them again (see Exercise 6.6).

6.3 Discussion and Bibliographic Notes

Reactive acting. BDI (Belief-Desire-Intention) architectures are reactive systems somewhat similar to the algorithms in Sections 6.1 and 6.2, but they differ with respect to their representation of actions and methods [276, 100, 275]. Most BDI systems will not replan, though there are a few exceptions [981, 1204]; and they will select and execute an untried method when failure occurs.

The Icarus algorithm [681] learns hierarchical logic programs that are analogous to

```
Incremental-Repair-2($\Sigma, Tree, v$)
    $s \leftarrow$ observed current state
    $p \leftarrow parent(v)$
    $\langle v_1, \dots, v_k \rangle \leftarrow Children(p)$
    $i \leftarrow$ the index of $v$ in $\langle v_1, \dots, v_k \rangle$, so that $v = v_i$
    for $j \leftarrow i$ to $k$ do
        $\pi_{\text{pending}} \leftarrow$ the sequence of actions after $v_j$ in $Tree$
        $Tree_1 \leftarrow$ Lookahead-RT($\Sigma, s, \langle content(v_i), \dots, content(v_j) \rangle$)
        if $Tree_1 \neq$ failure then
            $\pi_1 \leftarrow$ the sequence of actions in $Tree_1$'s leaf nodes
            if $\pi_{\text{pending}}$ is applicable in $\gamma(s, \pi_1)$ then
                $v_{\text{first}} \leftarrow$ the first action node in $Tree_1$
                in $Tree$, replace $v_i, \dots, v_j$ with $Children(root(Tree_1))$
                return $v_{\text{first}}$
    else
        if $parent(v)$ is the root of $Tree$ then return failure
        $v' \leftarrow$ the lowest ancestor of $v$ that is a task node
        return Incremental-Repair-2($\Sigma, Tree, v'$)
```

Algorithm 6.6. Incremental-Repair-2, a finer-grained repair algorithm. It first tries to replan v. If this fails, it tries repeatedly to replan v and some of its siblings, increasing the number of siblings each time. If that fails, it calls itself recursively on the task node above v.

HTN methods, and uses them for reactive acting. We discuss it further in Section 7.4.

Plan repair. In general, plan stability (see Section 3.5) can be accomplished more effectively for HTN planning than for classical planning, by using the HTN plan structure to localize the errors and failures, and using repair knowledge encoded into the HTN methods [1170].

In plan-space HTN planning, SIPE [1169] used a collection of "replanning actions" for repairing several kinds of errors. In PRIAR [577], validation graphs were used to identify disruptions and make patches to plan-space plans. The repair algorithm in [141] works by retracting the planning steps at the failure points, and then following the previous generation process as closely as possible.

The simple approach used in our Incremental-Repair and Incremental-Repair-2 algorithms is similar to that in [1155, 80]. Repairs can be done more effectively by reasoning about causal dependencies among actions, either analytically [64, 141, 436] or using simulation techniques [1218].

In [516], plan repair is done by modifying the planning domain to incorporate the observed outcome of an action failure, and recreating the refinement tree in the modified domain. This approach is appealing theoretically, but it excludes some intuitively plausible repairs that other repair algorithms can perform at higher levels of the refinement tree [432].

Some other approaches to plan repair include an HTN version of [141] gives a plan-repair algorithm for HTN plan-space plans that is similar to the classical plan-space repair algorithm in Section 5.2.2.

6.4 Exercises

```
Stack-Blocks($s_0, g$)
    $\pi \leftarrow \langle\rangle$
    while one or more blocks need to be moved do
        $C \leftarrow$ {blocks that are clear and need to be moved}
        $M \leftarrow$ {blocks for which the goal location is either the table or a block
          that doesn't need to be moved}
        if $C \cap M \neq \varnothing$ then
            choose a block in $C \cap M$ and move it to its goal location
        else choose a block in $C$ and move it to the table
    return $\pi$
```

Algorithm 6.7. Stack-Blocks, a blocks-world acting algorithm that finds near-optimal solutions.

6.1. Consider blocks-world problems in which the initial state includes the atom holding = nil, and the goal g is a set of loc atoms like those in Figure 2.8. Algorithm 6.7, Stack-Blocks, can find near-optimal solutions for such problems, where "optimal" means the smallest number of actions. Here are some terms used in the pseudocode:

- A block b is *clear* if top(b) = nil.
- If the goal g contains an atom of the form loc$(b) = c$, then c is block b's *goal location*. Otherwise b has no goal location.
- A block b *needs to be moved* if either b has a goal location that differs from its current location, or b's current location is a block that needs to be moved. Otherwise b doesn't need to be moved.

Answer the following questions:

(a) What sequence of actions will Stack-Blocks produce for the planning problem in Exercise 2.4(b)?

(b) Write a set of total-order HTN methods that encode Stack-Blocks. Assume there is a function *need-to-move*(b) that returns True if b needs to be moved and False otherwise, that you can use in the methods' preconditions.

6.2. In Figure 2.8, suppose the first "move" action drops the block onto the table, and all subsequent "move" actions operate correctly. For each of the following acting algorithms, tell what sequence of actions will be performed using the methods you wrote in Exercise 6.1.

(a) HTN-Run-Lookahead, with *Lookahead* = TO-HTN-Forward.
(b) HTN-Run-Lazy-Lookahead, with *Lookahead* = TO-HTN-Forward.
(c) HTN-Run-Repair, with *Lookahead-RT* = TO-HTN-Forward-RT and *HTN-Repair* = Incremental-Repair.
(d) HTN-Run-Repair, with *Lookahead-RT* = TO-HTN-Forward-RT and *HTN-Repair* = Incremental-Repair-2.

6.3. Repeat Exercise 6.2, but this time assume that for every block b, the first "move" operation on b drops it onto the table, and all subsequent "move" operations on b operate correctly.

6.4. In Exercise 6.2, suppose every "move" operation drops the block onto the table. For each of the four acting procedures in Exercise 6.2, will it eventually terminate or will it keep trying to move blocks forever?

6.5. In Example 6.3, can situations occur where Incremental-Repair and Incremental-Repair-2 will repeatedly return the same plan? If so, describe such a situation. If not, then explain why not.

6.6. Modify Incremental-Repair and Incremental-Repair-2 so that at each node of the refinement tree they will keep a list of the methods that they have already tried at that node, and will not try those methods again if called again at that node.

6.7. Repeat parts (c) and (d) of Exercises 6.2–6.5 using the modified versions of Incremental-Repair and Incremental-Repair-2 that you wrote in Exercise 6.6.

7 Learning HTN Methods

HTN planning algorithms require a set of HTN methods that provide knowledge about potential problem-solving strategies. Typically these methods are written by a domain expert, but this chapter is about some ways to learn HTN methods from examples. For simplicity, we focus specifically on how to learn total-order HTN methods.

This chapter is organized as follows. Section 7.1 describes a learning-by-demonstration problem in which a learner is given examples of plans to accomplish various tasks, and the objective is to learn HTN methods. Section 7.2 describes a more difficult version of the same problem: the examples consist only of plans, and the learner needs to use other information to infer what tasks the plans accomplish. Section 7.3 speculates briefly about prospects for a "planning-to-learn" approach in which a learner generates its own examples using a classical planner.

7.1 Learning Methods from Examples

Given a specification of a classical planning domain, suppose we want to learn a set of total-order HTN methods for various tasks from examples of how to accomplish the tasks. These examples could be provided by a tutor, or could be produced by observing a system in action, or if all of the tasks are goal tasks then the learner could generate the examples using a classical planner. More specifically, we will define a *solution example* to be a triple

$$e = (\mathrm{task}(e), \mathrm{pre}(e), \mathrm{plan}(e)), \tag{7.1}$$

where $\mathrm{pre}(e)$ is a set of literals that a state s must satisfy for $\mathrm{plan}(e)$ to accomplish $\mathrm{task}(e)$.[1] In this section we will define an algorithm to learn total-order HTN methods from solution examples.

7.1.1 Preliminaries

Before presenting the algorithm, we first need some definitions.

To reason about solution examples, we will need to extend γ to sets of literals.[2] If π is a plan and L is a set of literals that satisfies $\mathrm{pre}(\pi)$, then π is applicable in every state that satisfies L. If we call that set of states S_L, then we can define $\gamma(S_L, \pi) = \{\gamma(s, \pi) \mid s \in S_L\}$. It follows from Equation 2.15 that $\gamma(S_L, \pi)$ is the set of all states that satisfy the following set of literals:

[1] In this definition, we could have used a single initial state instead of $\mathrm{pre}(e)$. However, $\mathrm{pre}(e)$ is more general and will be useful in some of our computations.

[2] Some readers might find it helpful to think of this as an inverse of γ^{-1}, which produces a set of states to which γ can be applied.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

$$\{\text{an assignment } x = w \text{ for each effect } x \leftarrow w \text{ in eff}(a)\} \cup \{\text{every literal } x = w \text{ or } x \neq w \text{ in } L \text{ such that eff}(a) \text{ does not assign a value to } x\}. \quad (7.2)$$

With some mild abuse of notation, we will call this set of states $\gamma(L, \pi)$.

The learning algorithm will use an intermediate data structure that we will call a *method proposal*, which is a tuple of the form

$$\lambda = (\text{task}(\lambda), \text{pre}(\lambda), \text{subtasks}(\lambda), \text{plan}(\lambda)). \quad (7.3)$$

Here, the intent is to describe a potential method—except for the method's head, which will be added later. The three elements task(λ), pre(λ), and plan(λ)) are essentially a solution example: plan(λ) is a plan that is applicable and accomplishes task(λ) in any state that satisfies pre(λ). The other element, subtasks(λ), gives the proposed method's subtasks. Thus the proposal is to create a method to refine task(λ) into subtasks(λ) in any state that satisfies pre(λ), with a requirement that the algorithm will need to create other methods that refine subtasks(λ) into plan(λ).

Example 7.1. From Example 5.7 and Figure 5.2 we can get several examples of method proposals. The task is $t_1 = \{\mathsf{pile(c1) = p2}\}$, and the plan to accomplish it is the sequence of actions at the tree's leaf nodes, $\pi_1 = \langle a_1, a_2, a_3\rangle$. Let

$$g_0 = \gamma^{-1}(t_1, \pi_1) \quad (7.4)$$

as calculated in Exercise 7.1(a). Then t_1 can be accomplished in any state that satisfies g_0. Here are three different sets of method proposals to achieve it:

1. Here is a method proposal to refine t_1 directly into π_1:

$$\lambda_1 = (t_1, g_0, \pi_1, \pi_1). \quad (7.5)$$

2. Here are method proposals to refine t_1 into $\langle t_2, t_3, a_3\rangle$ as m_1 does in Figure 5.2, and to refine t_2 and t_3 into a_1 and a_2. We will not give the values of pre(a_1) and pre(a_2) here, but they can be calculated from the action schemas in Example 2.8.

$$\lambda_1' = (t_1, g_0, \langle t_2, t_3, a_3\rangle, \pi_1), \quad (7.6)$$
$$\lambda_2 = (t_2, \text{pre}(a_1), \langle a_1\rangle, \langle a_1\rangle), \quad (7.7)$$
$$\lambda_3 = (t_3, \text{pre}(a_2), \langle a_2\rangle, \langle a_2\rangle). \quad (7.8)$$

3. Here are method proposals for the refinements that $m_1, \ldots, m_5$ do in Figure 5.2. The values of pre(m_1), ..., pre(m_5) can be calculated from the action schemas in Example 2.8 and the method definitions in Example 5.4.

$$\lambda_1' = (t_1, \text{pre}(m_1), \langle t_2, t_3, a_3\rangle, \pi_1),$$
$$\lambda_2' = (t_2, \text{pre}(m_2), \langle t_4, t_5, a_1\rangle, \langle a_1\rangle),$$
$$\lambda_3 = (t_3, \text{pre}(m_3), \langle a_2\rangle, \langle a_2\rangle),$$
$$\lambda_4 = (t_4, \text{pre}(m_4), \langle\rangle, \langle\rangle),$$
$$\lambda_5 = (t_5, \text{pre}(m_5), \langle\rangle, \langle\rangle).$$

□

```
Methods-from-Examples($\Sigma_c, E$)
1  $Proposals \leftarrow \{(t, pre, \pi, \pi) \mid (t, pre, \pi) \in E\}$
   // Step 1: replace subplans with subtasks
2  foreach $\lambda \in Proposals$ do
       let $\langle t_1, \ldots, t_n \rangle$ be the sequence of tasks in subtasks($\lambda$)
3      while there is a $\lambda' \in Proposals$ such that subtasks($\lambda'$) matches a
          (contiguous) subsequence $\langle t_i, \ldots, t_j \rangle$ of $\langle t_1, \ldots, t_n \rangle$ and
          $t_1, \ldots, t_{i-1}$ are actions and $\gamma(\mathrm{pre}(\lambda), \langle t_1, \ldots, t_{i-1} \rangle) \models \mathrm{pre}(\lambda')$
        do
4          choose any such $\lambda'$ arbitrarily
           in subtasks($\lambda$), replace $\langle t_i, \ldots, t_j \rangle$ with task($\lambda'$)
   // Step 2: lift the method proposals
5  $\Omega \leftarrow \Sigma_c$'s ontology of typed objects
6  foreach $\lambda \in Proposals$ do
7      foreach constant symbol $b$ in $\lambda$ that is not a special constant do
           $R \leftarrow$ the set in $\Omega$ that contains $b$
           $x_b \leftarrow$ a new variable name with $Range(x_b) = R$
           in $\lambda$, substitute $x_b$ for every occurrence of $b$
   // Step 3: remove subsumed method proposals, then convert proposals to methods
8  foreach $\lambda, \lambda' \in Proposals$ do
       if $\lambda'$ is an instance of $\lambda$ then remove $\lambda'$ from $Proposals$
9  foreach $\lambda \in Proposals$ do
       $head \leftarrow$ a new method name and a list of the object variables in $\lambda$
       $\mathcal{M} \leftarrow \mathcal{M} \cup (head, \mathrm{task}(\lambda), \mathrm{pre}(\lambda), \mathrm{subtasks}(\lambda))$
   return $\mathcal{M}$
```

Algorithm 7.1. Methods-from-Examples, which learns methods from solution examples. Line 3 requires a γ function that has been extended to ground preconditions (see Equation 7.1).

7.1.2 The Learning Algorithm

Algorithm 7.1, Methods-from-Examples, learns total-order HTN methods from a domain Σ_c and a set E of solution examples. Its objective is to create a set of methods $\mathcal{M}$ such that for every example (t, pre, π) and every state s that satisfies pre, π is a solution for the planning problem $((\Sigma_c, \mathcal{M}), s, t)$.

In Line 1, the algorithm initializes *Proposals* so that for each example in E there is a method proposal like the one in Equation 7.5: each subtask list is itself the desired solution. Although *Proposals* could be converted directly into a set of methods, they would be unable to solve any planning problems other than the ones in E. To make the methods more general, so the algorithm performs the following steps.

Step 1: At Line 2, the algorithm modifies the methods to generate subtasks that the other methods can accomplish. For each $\lambda, \lambda' \in Proposals$ such that subtasks(λ')

matches part of subtasks(λ), the algorithm modifies λ so that instead of generating subtasks(λ) directly, it will use task(λ') as an intermediate step.

Step 2: At Line 6, the algorithm makes the method proposals more general by *lifting* them, that is, replacing object constants with variables.[3] This assumes that the ontology Ω in Line 5 includes a set of *special constants* that should not be lifted, such as T, F, and nil.[4]

Step 3: Once the method proposals are lifted, some of them may subsume others. At Line 8, the algorithm removes the subsumed ones from *Proposals*. At Line 9, it converts the remaining ones into methods and adds them to $\mathcal{M}$. Then it returns $\mathcal{M}$.

Example 7.2. Continuing from Example 7.1, let us consider a call to Methods-from-Examples(Σ_c, $\{e_1, e_2, e_3\}$), where

$$e_1 = (t_1, g_0, \pi_1), \quad e_2 = (t_2, \text{pre}(a_1), \langle a_1 \rangle), \quad e_3 = (t_3, \text{pre}(a_2), \langle a_2 \rangle).$$

For these three examples, at Line 1 the algorithm generates the method proposals λ_1, λ_2, and λ_3 in Equations 7.5, 7.7, and 7.8, and puts them into *Proposals*.

Step 1 modifies λ_1 by replacing a_1 and a_2 with t_2 and t_3, which makes λ_1 identical to λ'_1 in Equation 7.6. Step 2 lifts the proposals by replacing r1, c1, c2, d1, d2, p1, and p2 with $r_1 \in \mathit{Robots}$, $c_1, c_2 \in \mathit{Containers}$, $d_1, d_2 \in \mathit{Docks}$, and $p_1, p_2 \in \mathit{Piles}$. Afterwards, *Proposals* contains:

$$\begin{aligned}
\hat{\lambda}_1 &= (\hat{t}_0, \hat{g}_0, \langle \hat{t}_1, \hat{t}_2, \hat{a}_3 \rangle, \langle \hat{a}_1, \hat{a}_2, \hat{a}_3 \rangle),\\
\hat{\lambda}_2 &= (\hat{t}_1, \text{pre}(\hat{a}_1), \langle \hat{a}_1 \rangle, \langle \hat{a}_1 \rangle),\\
\hat{\lambda}_3 &= (\hat{t}_2, \text{pre}(\hat{a}_2), \langle \hat{a}_2 \rangle, \langle \hat{a}_2 \rangle),
\end{aligned}$$

where

$$\begin{aligned}
&\hat{t}_0 = \{\text{pile}(c_1) = p_2\}, && \hat{t}_1 = \text{get-container}(r_1, c_1), && \hat{t}_2 = \text{navigate}(r_1, d_2),\\
&\hat{a}_3 = \text{put}(r_1, c_1, c_3, p_2, d_2), && \hat{a}_1 = \text{take}(r_1, c_1, c_2, p_1, d_1), && \hat{a}_2 = \text{move}(r_1, d_1, d_2).
\end{aligned}$$

For brevity, the values of $\hat{g}_0$, pre($\hat{a}_1$), and pre($\hat{a}_2$) are not shown here, but they can be calculated from the action schemas and method definitions in Examples 2.8 and 5.4.

In Step 3, the loop at Line 8 does not change *Proposals* because no candidate subsumes any of the others. The loop at Line 9 converts each $\hat{\lambda}_i$ into a method and puts it into $\mathcal{M}$. For example, it converts $\hat{\lambda}_3$ to the following method (the method and variable names may differ from those shown here):

```
mu3($r_1, d_1, d_2$)
  task: navigate($r_1, d_2$)
   pre: adjacent($d_1, d_2$), loc($r_1$) = $d_1$, occupied($d_2$) = F
   sub: move($r_1, d_1, d_2$)
```

This is like m2-navigate in Example 5.4, but with the additional precondition occupied(d_2) = F.

Finally, Methods-from-Examples returns $\mathcal{M}$. □

[3] An unresolved issue is how far to lift an object constant if the ontology Ω organizes them into multiple levels. For example, if c1 $\in$ *Containers* $\subset$ *Positions*, then should c1 be replaced with a variable whose range is *Containers* or one whose range is *Positions*?

[4] Section 4.2.1 will also use a set of special constants for this purpose.

7.1.3 Properties of the Algorithm

Completeness. It is not hard to show that Methods-from-Examples is complete with respect to E, that is, it will produce a set of methods $\mathcal{M}$ capable of solving all of the planning problems in E. In other words, for each example $(t, pre, \pi) \in E$ and for every state s that satisfies pre, π is a solution for the planning problem $((\Sigma_c, \mathcal{M}), s, t)$. The proof is by induction on the number of refinement steps.

In many cases, E may be just a subset of a much larger (and possibly infinite) set of examples E'. In such cases, Methods-from-Examples is *asymptotically* complete. For $i = 0, 1, 2, \ldots$, let E_i be the set of all examples (t, pre, π) such that $\text{length}(\pi) \leq i$. Let $\mathcal{M}' = \bigcup_i \mathcal{M}_i$, where $\mathcal{M}_i$ is the set of methods returned by Methods-from-Examples(Σ_c, E_i). Then for every example $(t, pre, \pi) \in E'$ and every state s that satisfies pre, π is a solution for the planning problem $((\Sigma_c, \mathcal{M}'), s, t)$.

Complexity. It is easy to see that Methods-from-Examples has low-order polynomial running time. The hardest parts of the computation are the nested loops at lines 2 and 9, which compare the subtask lists of every pair of proposals, for a total of $O(p^2)$ comparisons where p is the number of proposals. Each comparison is a string-matching problem that takes linear time in the length of the subtask lists, which is $O(n)$ where $n = |\pi|$. Thus the overall computational complexity is $O(p^2 n)$.

Minimality. Methods-from-Examples satisfies the following minimality property. At the end of Step 1, every proposal in *Proposals* has a minimal set of subtasks. For each $\lambda \in Proposals$, $\text{subtasks}(\lambda)$ cannot be made smaller, because none of its subsequences can be replaced with any of the tasks in $\{\text{task}(\lambda') \mid \lambda' \in Proposals\}$.

At the end of Step 2, when all of the method proposals are lifted, some of them may now be able to produce parts of $\text{subtasks}(\hat{\lambda})$, so the minimality property no longer holds. However, if we modify the algorithm to repeat the loop at Line 1 again after the loop at Line 6 has finished, then the final set of method proposals—and thus the set of methods returned by the algorithm—will again have minimal sets of subtasks.

Soundness. Given a set of examples E, if Methods-from-Examples(Σ_c, E) returns a set of methods $\mathcal{M}$, then for every example $e \in E$ and every planning problem $P = ((\Sigma_c, \mathcal{M}), s, \text{task}(e))$ such that s satisfies $\text{pre}(e)$, P is solvable. If this is our definition of soundness, then Methods-from-Examples is sound.

However, suppose the examples in E were taken from an HTN planning domain $(\Sigma_c, \mathcal{M}')$, and we want Methods-from-Examples to learn a set of methods $\mathcal{M}$ that is equivalent to $\mathcal{M}'$. Then we might want to require that for every s and T, the planning problems $P = ((\Sigma_c, \mathcal{M}), s, T)$ and $P' = ((\Sigma_c, \mathcal{M}'), s, T)$ should have the same sets of solutions. If that is our definition of soundness, then Methods-from-Examples is not sound. To see why, recall that the methods in $\mathcal{M}$ are lifted. If some of the methods in $\mathcal{M}'$ are not lifted, then P may have more solutions than P'.

The problem here is not whether Methods-from-Examples is sound, it is that the examples in E do not completely specify $\mathcal{M}'$. The same set E could have been generated by $\mathcal{M}'$, or by the set $\mathcal{M}$ returned by Methods-from-Examples, or by some other sets of methods. The learning algorithm cannot tell which, because it operates offline with no way to generate additional examples.[5]

A possible fix for the problem might be to modify Methods-from-Examples to take both positive and negative examples, the latter being examples of solutions that the methods should never produce. With such a modification, a sufficiently large set of examples might be able to make the algorithm converge to a unique set of methods. However, this idea is quite speculative, and we know of no work on the topic.

Generality. In general, we would like the methods to be capable of solving many more planning problems than the ones in E. In the previous paragraphs we pointed out that it is difficult to specify what additional planning problems the methods in $\mathcal{M}$ should be able to solve. The following paragraphs give examples of some cases in which the methods in Example 7.2 can solve some planning problems not in E, and an example of a case in which they cannot.

In the methods produced by Methods-from-Examples, all of the objects have been replaced by object variables, so their ability to solve planning problems is unaffected by changes to the objects' names. It also is unaffected by changes to irrelevant domain features. For example, the methods learned in Example 7.2 can still achieve t_1 if we rename c1 and r1 to c17 and r28, or if we insert some additional containers under c3, or if we add some piles to the loading docks.

Unfortunately, if we put one or more containers on top of c1 then the methods cn no longer achieve t_1, because $\mathcal{M}$ doesn't contain the recursive m2-uncover method in Example 5.4. If we add examples to E in which there are containers on c1, then the algorithm will learn some methods to uncover c1, but it will not learn the recursion step at the end of m2-uncover. It will only learn methods to remove the number of containers in the examples. They will not be able to achieve t_1 if we put more containers onto c1 than are present in the examples.

As a work-around, one could modify an HTN planner to call a classical planner in such cases—but this would negate some of HTN planning's advantages, such as the ability for an HTN domain author to constrain the search space to make the planner avoid undesirable solutions and exit quickly when no desirable solution can be found. It would be more desirable to learn a simple set of methods that could perform unlimited recursion. Section 7.4 discusses some work on this topic, but more needs to be done.

Refinement trees. The set E does not need to be restricted to contain just solution examples. It may instead contain *task refinements* of the form

$$e = (\text{task}(e), \text{pre}(e), \text{subtasks}(e)), \tag{7.9}$$

where $\text{subtasks}(e)$ is a list of subtasks. These might be taken from a refinement tree, or might be provided by a human expert who has good ideas about what subtasks to use.

[5]A similar problem with offline learning of action schemas in Section 4.2.1 is addressed by online learning Section 4.3.

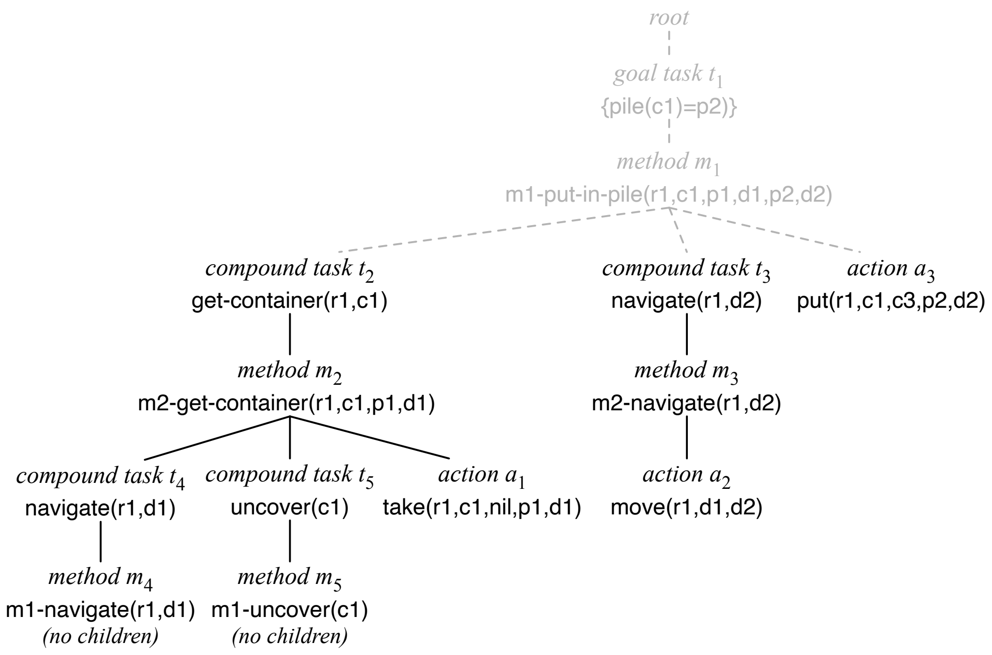


Figure 7.1. Three subtrees of the refinement tree in Figure 5.2, with the missing part of the refinement tree shown in gray.

If E includes all of the task refinements in a refinement tree, then this provides some very specific information about what the method proposals should be, and in this case Step 1 of Methods-from-Examples can be skipped entirely. For example, if E contains all of the task refinements shown in Figure 5.2, then this tells the algorithm to use the method proposals in part 3 of Example 7.1.

E might instead include just some of a solution tree's task refinements. In Example 7.2, if E were the task refinements shown in the three subtrees shown in Figure 7.1, this would tell Methods-from-Examples to use the method proposals λ_2', λ_3, λ_4, and λ_5 in part 3 of Example 7.1. If we also gave it the solution example $e_1 = (t_1, g_0, \pi_1)$ from Example 7.2, this would enable it to create λ_1'.

7.2 Learning Methods from Plans

Suppose we have a set of plans, an initial state for each plan, and some tasks for which we would like to learn methods. As before, the plans could be provided by a tutor, or produced by observing a system in action, or generated by the learner using a classical planner. If we have an easy way to infer whether each task has been accomplished, then we can create solution examples from which Methods-from-Examples can learn methods. This section presents an algorithm for doing that.

As a way to infer whether each task has been accomplished, let us suppose that for each task we have two *annotations*: one telling what conditions need to hold when the task begins, and one telling what conditions need to hold when the task finishes. For now we will assume that these annotations are provided by a human. However, it would be desirable to have a way to create them automatically, and some preliminary work has been done on this topic (this is discussed further in Section 7.4).

More specifically, an *annotated task*[6] is a triple $\tau = (\mathrm{task}(\tau), \mathrm{pre}(\tau), \mathrm{eff}(\tau))$, where

- $\mathrm{task}(\tau)$ is a compound task or goal task;
- $\mathrm{pre}(\tau)$ is a set of literals called τ's precondition, which should be true before accomplishing τ;
- $\mathrm{eff}(\tau)$ is a set of literals that must be true immediately after τ has been accomplished. Although we will call it τ's "effects," it is essentially a goal: any successful refinement of τ must produce a state in which $\mathrm{eff}(\tau)$ is true.

Algorithm 7.2, Methods-from-Plans, uses plans to learn methods for annotated tasks. Its input includes a classical planning domain, a set of pairs of plans and their initial states, and a set of annotated tasks. It works as follows.

```
Methods-from-Plans($\Sigma_c$, $Pairs$, $\mathcal{T}$)
    $E \leftarrow \varnothing$
1   foreach pair $(s_0, \pi) \in Pairs$ do
2       $\langle a_1, \ldots, a_n \rangle \leftarrow \pi$;  $\langle s_0, \ldots, s_n \rangle \leftarrow \widehat{\gamma}(s_0, \pi)$
3       foreach annotated task $\tau \in \mathcal{T}$ do
            foreach nonempty subplan $\langle a_i, \ldots, a_j \rangle$ of $\pi$ do
4               if $s_{i-1} \models \mathrm{pre}(\tau)$ and $s_{i-1} \not\models \mathrm{eff}(\tau)$ and $s_j \models \mathrm{eff}(\tau)$ then
5                   // The plan $\langle a_i, \ldots, a_j \rangle$ accomplishes $\tau$
                    $pre \leftarrow \mathrm{pre}(\tau) \cup \gamma^{-1}(\mathrm{eff}(\tau), \langle a_i, \ldots, a_j \rangle)$
6                   add $(\mathrm{task}(\tau), pre, \langle a_i, \ldots, a_j \rangle)$ to $E$
7       if $\exists s_j \in \{s_0, \ldots, s_n\}$ such that $s_j \models \mathrm{pre}(\tau)$ and $s_j \models \mathrm{eff}(\tau)$ then
            // The empty plan accomplishes $\tau$
            add $(\mathrm{task}(\tau), \mathrm{pre}(\tau) \cup \mathrm{eff}(\tau), \langle\rangle)$ to $E$
8   foreach $\tau, \tau'$ in E do
        if $\tau'$ is an instance of $\tau$ then remove $\tau'$ from $E$
9   return Methods-from-Examples($\Sigma_c$, $E$)
```

Algorithm 7.2. Methods-from-Plans, which learns methods for annotated tasks.

The outer loop at Line 1 iterates through each pair (s_0, π). From the initial state s_0 and the actions in π, Line 2 calculates the sequence of states that the plan produces.

The nested loops at Line 3 compare each annotated task τ with each nonempty subplan of π.[7] If a subplan's starting and ending states satisfy $\mathrm{pre}(\tau)$ and $\mathrm{eff}(\tau)$, respectively, then Line 6 adds to E a solution example saying that the subplan accomplishes τ. If a state s_i satisfies both $\mathrm{pre}(\tau)$ and $\mathrm{eff}(\tau)$, then Line 7 adds to E a solution

[6]Syntactically, annotated tasks are identical to the abstract actions in Section 5.5.1. However, the purpose here is not to define an abstract action, but instead to aid the learning algorithm by providing information about what the desired HTN methods for $\mathrm{task}(\tau)$ should do.

[7]As in the complexity analysis of Methods-from-Examples, it is not hard to see that the time complexity is low-order polynomial.

example saying that the empty plan accomplishes τ. This is for creating methods with no subtasks, such as m1-get-container and m1-uncover in Example 5.4.

The loop at Line 8 removes from E any examples that are subsumed by others. Finally, Line 9 calls Methods-from-Examples on E, and returns the resulting set of methods.

Example 7.3. The following annotated tasks correspond to t_1, t_2, and t_3 in Figure 5.2.

$$\begin{aligned}
\tau_1 &= (t_1, && \{\text{pile(c1)} = \text{p1}\}, && \{\text{top(p2)} = \text{c1}\}),\\
\tau_2 &= (\text{get-container(r1,c1)}, && \{\text{cargo(r1)} = \text{nil}\}, && \{\text{cargo(r1)} = \text{c1}\}),\\
\tau_3 &= (\text{navigate(r1,d2)}, && \{\text{loc(r1)} = \text{d1}\}, && \{\text{loc(r1)} = \text{d2}\}).
\end{aligned}$$

Suppose we call Methods-from-Plans$(\Sigma_c, \{(s_0, \pi_1)\}, \{\tau_1, \tau_2, \tau_3\})$, where Σ_c, s_0, and π_1 are as in Examples 7.1 and 7.2.

In the first iteration of the loop at Line 3, $\tau = \tau_1$, and the only subplan of π_1 that satisfies Line 4 is π_1 itself. In Line 5,

$$pre = \text{pre}(\tau) \cup \gamma^{-1}(\text{eff}(\tau_1), \pi_1) = \text{pre}(\tau) \cup \gamma^{-1}(t_1, \pi_1) = \gamma^{-1}(t_1, \pi_1) = g_0,$$

so Line 6 will add (t_1, g_0, π_1) to E, which is the same as e_1 in Example 7.2.

In the second iteration of the loop, $\tau = \tau_2$, and the subplans of π_1 that accomplish τ_2 are $\langle a_1 \rangle$ and $\langle a_1, a_2 \rangle$, so Line 6 will add to E the following two solution examples, the first of which is the same as e_2 in Example 7.2:

$$(t_2, \text{pre}(\tau_2) \cup \gamma^{-1}(\text{eff}(\tau_2), \langle a_1 \rangle), \langle a_1 \rangle) = (t_2, \text{pre}(a_1), \langle a_1 \rangle),$$
$$(t_2, \text{pre}(\tau_2) \cup \gamma^{-1}(\text{eff}(\tau_2), \langle a_1, a_2 \rangle), \langle a_1, a_2 \rangle) = (t_2, \text{pre}(a_1) \cup \text{pre}(a_2), \langle a_1, a_2 \rangle).$$

In the third iteration of the loop, $\tau = \tau_3$, and there are four subplans of π_1 that accomplish τ_2: $\langle a_2 \rangle$, $\langle a_1, a_2 \rangle$, $\langle a_2, a_3 \rangle$, and π_1. Line 6 will add to E a solution example for each of them.

From the seven examples, Methods-from-Examples will create seven methods. Depending on what choices it makes in Line 4, these may or may not include the methods in Example 7.2. Methods different from the ones in Example 7.2 are unlikely to be very useful, and thus good heuristics are needed to guide the choices in Line 4. Research is needed on this topic. □

If we know the task hierarchy in advance, then it can be used to optimize the learning, by ordering the examples in E in a bottom-up fashion, starting with the bottom-level tasks and going the top-level ones. This way, the methods learned for each task τ can be learned from the methods already learned for its subtasks, constructing the refinement tree as we go. As described at the end of Section 7.1.3 under the topic of refinement trees, this would allow Step 1 of Methods-from-Examples to be skipped. The same approach can be used even if we do not have E, provided that we have a classical planner to generate plans for the annotated tasks.

7.3 Planning to Learn

In principle, the method-learning algorithms in this chapter could be used as part of a planning-to-learn approach in which a learner uses the learning algorithms in the previous sections, along with a classical planner from which to generate examples. The idea would be to repeatedly create planning problems, use the planner to solve them, give the planner's solutions as input to the learning algorithm, and use the learning algorithm's output to decide what planning problems to generate next. The learner could strategically call the planner on planning problems for which the solutions would provide information about how to construct new methods, or information about conditions under which one of its methods does or doesn't work.

One way to do this would be to start with a set of annotated tasks. For an annotated task $\tau = (\text{task}(\tau), \text{pre}(\tau), \text{eff}(\tau))$, the learner could call a classical planner on a planning problem in which the initial state satisfies $\text{pre}(\tau)$ and the goal is $\text{eff}(\tau)$. The resulting plan would then provide a solution example that could be given as input to Methods-from-Plans. If Methods-from-Plans were modified to work incrementally, then the learner could make strategic choices of which annotated tasks to use as input to the classical planner, as discussed in the previous paragraph.

If no tasks were available to start from, another possibility would be to use a landmark algorithm (see Section 3.2.3) to create planning problems, return a sequence of landmarks for each planning problem, and use the landmarks to divide the classical planner's solutions into collections of examples to use as input to Methods-from-Examples. In this case, the landmarks would constitute the tasks. This could be done incrementally as described in the previous paragraph.

No work has yet been done on these approaches, and it is unclear how well either of them would work in practice. One challenge would be how to guide the generation of new planning problems to solve. Another would be how to prevent the generation of a large number of methods of which very few are useful. If ways can be found to address these challenges, a next step might be to build an actor that integrates the operation of the learner, an acting algorithm, and an HTN planning algorithm.

7.4 Discussion and Bibliographic Notes

Learning total-order HTN methods. Most HTN method-learning algorithms are for total-order HTN methods. The best-known of these is HTN-Maker [511]. Instead of making a set of examples like Methods-from-Plans does, HTN-Maker goes directly into a computation to produce methods. It makes choices deterministically, going backwards from the goal. Despite these differences, HTN-Maker provided the inspiration for Methods-from-Plans.

CurricuLearn [710] is a modified version of HTN-Maker that learns from a *curriculum* [112], a sequence of examples of increasing difficulty that guide the learner to (in this case) first learn methods for small tasks, then larger methods that build on the smaller ones. Methods-from-Examples was inspired by CurricuLearn, though the algorithms themselves are different.

Another approach [498] is to learn methods that contain task names without any parameters for the tasks, and then add the parameters. To learn the methods, the author uses a simplified version of HTN-Maker that ignores all of the action parameters, together with a collection of preprocessing and postprocessing algorithms to optimize the set of methods that are learned. To decide which parameters to add to the methods, the author describes algorithms to generate a set of candidates and then use MAX-SMT optimization to choose which parameters to use.

HTNLearn [1234] is designed to learn definitions of both methods and actions. Its input includes plans that are augmented with a sequence of subtrees like those in Figure 7.1, along with partial information about the intermediate states between the leaf nodes of the refinement trees. By compiling statistics on the atoms in the intermediate states, the algorithm creates weighted hypotheses about the actions and methods. It feeds this information, along with various other constraints, into a weighted MAX-SAT solver. The solver's solution provides preconditions for each method, and preconditions and effects for each action.

At the end of the "generality" topic in Section 7.1.3, we discussed the need for a way to learn recursive methods. To date, two approaches have been proposed that can do this in some situations [498, 710], but this is otherwise an open problem.

Near the beginning of Section 7.2, we mentioned the desirability of having an algorithm to create task annotations automatically. One of the results in [710] is an algorithm to create annotations from landmarks. It works well in several test cases, but more work remains to be done on this topic.

Learning partial-order HTN methods. Learning partial-order HTN methods is more complicated, but there are a few works on the topic.

The work described in [738] uses a technique called invariance analysis. Given a classical planning domain and a problem in that domain, the idea is to first construct one or more *invariants*, each of which is a set of atoms such that exactly one of the atoms is true at every state in the domain. For each invariant, the algorithm constructs an *invariant graph* showing the possible transitions from each atom to the others. For example, if c1 is a container in a DWR problem, then the set of possible values of pos(c1) is an invariant. Two of the possible transitions might be from pos(c1) = loc1 to pos(c1) = r1, and vice versa. The learning algorithm uses paths in the invariant graphs to construct methods for achieving desired values.

The MethodRefine algorithm [1186] is for situations where some of the methods' bodies are incompletely specified, that is, they include some but not all of the subtasks that are needed to solve a planning problem. The objective is to find what additional subtasks are needed, and modify the method to include them. To do this, the authors use a hybrid HTN/classical planner (see Section 5.3) to solve the planning problem, and augment the methods to include the additional tasks that appear in the hybrid planner's solution.

Other related work. Icarus [681] learns hierarchical logic programs that are analogous to HTN methods, and uses them for reactive acting. Its input includes a planning problem, a set of *primitive skills* that are similar to actions, and a hierarchy of *concepts* that are abstract conditions along with ways to infer whether those conditions hold. Given a planning problem, it uses a variant of backward search to solve the problem. When it recognizes places in its solution plan where the various concepts hold, it can use those parts of the plan to form nonprimitive skills that are analogous to methods.

HPNL [680] includes a representation and a learning algorithm that improve on the ones in Icarus, and a planner that uses them. The primary improvement is a way for the methods to include information that conditions their applicability on what other goals the planner needs to achieve.

7.5 Exercises

7.1. In the following calculations, use the action schemas in Example 2.8, the definition of γ^{-1} in Equation 3.15, and the method definitions in Example 5.4:

(a) Calculate $\gamma^{-1}(t_1, \pi_1)$ in Example 7.1.
(b) Calculate $\text{pre}(m_1), \ldots, \text{pre}(m_5)$ in Example 7.1.
(c) Calculate $\text{pre}(\hat{a}_1)$, $\text{pre}(\hat{a}_2)$, $\text{pre}(\hat{a}_3)$, and $\hat{g}_0$ in Example 7.2.

7.2. Prove that if g is a ground set of literals and $\gamma^{-1}(g, a)$ is defined, then it is also a ground set of literals. Hint: the proof is by induction on the length of π.

7.3. Modify Methods-from-Examples to remove the requirement in Line 3 that $t_1, \ldots, t_{i-1}$ be actions. The modification is rather complicated, and will introduce a lot of computational overhead. It involves the following steps:

- Modify TO-HTN-Forward to work on partial states.
- In Line 3, replace $\gamma(\text{pre}(\lambda), \langle t_1, \ldots, t_{i-1}\rangle$ with $\gamma(\text{pre}(\lambda), \pi)$, where π is the plan returned by TO-HTN-Forward$(\text{pre}(\lambda), \langle t_1, \ldots, t_{i-1}\rangle)$.
- To provide the methods that TO-HTN-Forward will need, use a copy of the loop at Line 9 of Methods-from-Examples.

7.4. For choosing λ' in Line 4 of Methods-from-Examples, some possible heuristics are to choose a λ' that maximizes j, or i, or $j - i$, or some combination of them. Another possibility is to choose a set of method proposals that produce non-overlapping portions of *sub*, in a way that maximizes the total length of these portions. Compare these heuristics on several examples. For each of them, try to come up with an example in which it does better than the others.

7.5. In Example 7.2, write the methods that the algorithm produces from $\hat{\lambda}_1$ and $\hat{\lambda}_2$.

7.6. Let Σ_c be the simple DWR domain illustrated in in Figure 7.2. The loading dock and crane do not have names. The ontology of typed objects is

$$\mathit{Objects} = \mathit{Containers} \cup \mathit{Piles} \cup \mathit{Positions} \cup \mathit{Special};$$
$$\mathit{Containers} = \{\text{c1}, \text{c2}, \text{c3}, \text{c4}, \text{c5}\}; \qquad \mathit{Piles} = \{\text{p1}, \text{p2}, \text{p3}\};$$
$$\mathit{Positions} = \mathit{Containers} \cup \{\text{nil}\}; \qquad \mathit{Special} = \{\text{nil}\}.$$

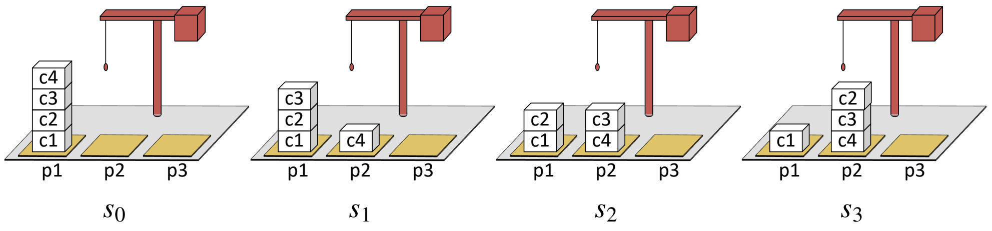


Figure 7.2. Four states for Example 7.2.

There are two action schemas, where $c \in \textit{Containers}$, $c' \in \textit{Positions}$, $p \in \textit{Piles}$:

$$\begin{aligned}&\mathsf{take}(c, c', p) \qquad \textit{// take container c off of c' in pile p}\\&\quad \text{pre: } \mathsf{holding} = \mathsf{nil},\ \mathsf{pos}(c) = c',\ \mathsf{top}(p) = c\\&\quad \text{eff: } \mathsf{holding} \leftarrow c,\ \mathsf{pos}(c) \leftarrow \mathsf{nil},\ \mathsf{pile}(c) \leftarrow \mathsf{nil},\ \mathsf{top}(p) \leftarrow c'\end{aligned}$$

$$\begin{aligned}&\mathsf{put}(c, c', p) \qquad \textit{// put container c onto c' in pile p}\\&\quad \text{pre: } \mathsf{holding} = c,\ \mathsf{top}(p) = c'\\&\quad \text{eff: } \mathsf{holding} \leftarrow \mathsf{nil},\ \mathsf{pos}(c) \leftarrow c',\ \mathsf{pile}(c) \leftarrow p,\ \mathsf{top}(p) \leftarrow c\end{aligned}$$

The state variables have the following ranges:

$$\begin{aligned}\textit{Range}(\mathsf{pile}(c)) &= \textit{Piles} \cup \{\mathsf{nil}\};\\ \textit{Range}(\mathsf{holding}) = \textit{Range}(\mathsf{pos}(c)) &= \textit{Range}(\mathsf{top}(p)) = \textit{Positions}.\end{aligned}$$

Let s_0, s_1, s_2, and s_3 be the states shown in Figure 7.2, and let

$$\begin{aligned}\pi_0 &= \langle\rangle,\\ \pi_1 &= \langle \text{take(c2,c1,p1), put(c2,c3,p2)} \rangle,\\ \pi_2 &= \langle \text{take(c3,c2,p1), put(c3,c4,p2)} \rangle \cdot \pi_1,\\ \pi_3 &= \langle \text{take(c4,c3,p1), put(c4,nil,p2)} \rangle \cdot \pi_2.\end{aligned}$$

Notice that $\gamma(s_3, \pi_0) = \gamma(s_2, \pi_1) = \gamma(s_1, \pi_2) = \gamma(s_0, \pi_3) = s_3$.

Suppose we call Methods-from-Examples(Σ_c, E), where

$$E = \{(t, s_3, \pi_0), (t, s_2, \pi_1), (t, s_1, \pi_2), (t, s_0, \pi_3)\},$$

and where t = make-clear(c1, p1) is the task of removing the containers above c1 in p1. Answer the following questions:

(a) What method proposals will be created in Line 1?
(b) What will the method proposals be at the end of the loop at Line 1?
(c) What will the lifted method proposals be at the end of the loop at Line 6?
(d) In the loop at Line 8, what method proposals, if any, will be removed?
(e) What methods will be returned?
(f) How would your answers change if *Special* were empty?

7.7. Prove the completeness result in the first paragraph of Section 7.1.3.

7.8. Modify Methods-from-Examples to make it capable of taking as input the examples produced by Make-Examples in Example 7.3. Run the modified algorithm by hand, to demonstrate that it will create seven methods as mentioned at the end of Example 7.3.

Part III

Probabilistic Models

One may even say, strictly speaking, that almost all our knowledge is only probable.

Pierre-Simon de Laplace, *Essai philosophique sur les probabilités*, 1814

The motivations for acting and planning with probabilistic models are about handling uncertainty in a quantitative way, with optimal or near optimal decisions.

The future is never entirely and precisely predictable. Uncertainty can be due to exogenous events in the environment, from nature and unmodeled actors, to noisy sensing and information gathering actions, to possible failures and outcome of imprecise or intrinsically nondeterministic actions (e.g., throwing a dice). Models are necessarily incomplete. Knowledge about open environments is partial. Part of what may happen can be only be modeled with uncertainty. Even in closed predictable environments, complete deterministic models may be too complex to develop.[8]

This part of the book explores approaches for using probabilistic models to handle the uncertainty and nondeterminism in acting, planning and learning. These approaches are based on optimization methods for Markov decision processes (MDP). Chapter 8 explains the basic principles and representations for MDP problems, starting with a simple flat representation, then considering a structured representation with domain decomposition and hierarchization methods. It addresses the issues of acting with and modeling a probabilistic domain. Chapter 9 is about planning techniques with probabilistic domains. Dynamic programming, heuristic search, linear programming, online and sampling algorithms for these problems are presented and analyzed. Chapter 10 considers reinforcement learning for probabilistic models.

[8]For example, the kinetic theory of gazes in statistical physics aggregates deterministic laws of elementary particles as statistical models.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

8 Probabilistic Representation and Acting

In probabilistic models, an action has several possible outcomes that are not equally likely; their distribution can be estimated, relying for example on statistics of past observations. The purpose is to act *optimally* with respect to an optimization criteria of the estimated likelihood of action effects and their cost.

The usual formal probabilistic models are *Markov decision processes* (MDPs). An MDP is a nondeterministic state-transition system with a probability distribution and a cost distribution. The probability distribution defines how likely it is to get to a state s' when an action a is performed in a state s.

This chapter presents MDPs in the flat then the structured state-space representations. Section 8.3 covers modeling issues of a probabilistic domain with MDPs and variants such as the Stochastic Shortest Path model (SSP) or the Constrained MDP (C-MDP) model. Section 8.4 focuses on acting with MDPs. Partially Observable MDPs and other extended models are discussed in Section 8.5.

8.1 Basic MDP Representation

This section introduces the main definitions and concepts needed for modeling a probabilistic domain with a flat representation.

A probabilistic state-transition system is said to be *Markovian* if the probability distribution of the next state depends only on the current state and not on the sequence of states that preceded it. The system is said to be *stationary* when the probability and the cost distributions remain invariant over time.[1]

Markovian and stationary properties are not intrinsic features of the world but are properties of its model. It is possible to take into account the dependence on the past within a Markovian description by defining an extended state that includes the current configuration of the world, as well as information about how the system has reached that configuration, as illustrated next.

Example 8.1. Consider a domain whose dynamic depends not only on the current value of a state variable x_t, but also on the past two values x_{t-1} and x_{t-2}. Let us add to the state space two state variables $x'_t = x_{t-1}$ and $x''_t = x_{t-2}$. A model with the three variables x, x', x'' is Markovian.

A similar idea can be used to handle time dependence to obtain a stationary model. Consider a city traffic domain whose model varies over time, e.g., rush hours, day/night time, days of the week. Adding a state variable ranging over these categories of traffic levels leads to a stationary model. □

[1]Note that a stationary system changes over time, but according to the same unvarying model.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

A probabilistic state-transition system is *observable* when the actor can always determine in which state it is. There are several families of observable stationary Markovian models, some with slight variations, e.g., transition rewards instead of costs). Other models consider significant extensions, such as partially observable states, or concurrent and durative actions. We focus here on observable stationary Markovian systems, and discuss extensions in Section 8.5.

8.1.1 Main Definitions

Definition 8.2. A *probabilistic domain* is a tuple $\Sigma = (S, A, \gamma, \Pr, \text{cost})$ where:

- S and A are finite sets of states and actions, respectively.
- $\gamma : S \times A \rightarrow 2^S$ is the state transition function; the states $s' \in \gamma(s, a)$ are distributed according to a probability distribution $\Pr(s'|s, a)$. The set of applicable actions in a state s is denoted $\mathit{Applicable}(s) = \{a \in A | \gamma(s, a) \neq \varnothing\}$.[2]
- $\Pr(s'|s, a)$ is the probability of reaching s' when action a takes place in s; $\Pr(s'|s, a) \neq 0$ iff $s' \in \gamma(s, a)$.
- $\text{cost} : S \times A \times S \rightarrow \mathbb{R}$; $\text{cost}(s, a, s')$ is the cost of a when reaching s' from s. The opposite of the cost is called the *reward*: $r(s, a, s') = -\text{cost}(s, a, s')$. $\square$

The actor wants to act in Σ following a plan expressed as a policy:

- A *policy* is a function $\pi : S' \rightarrow A$, with $S' \subseteq S$, such that for every $s \in S'$, $\pi(s) \in \mathit{Applicable}(s)$. Thus $\mathit{Domain}(\pi) = S'$.
- The *transitive closure* of s and π is the set of all states reachable from s using π. Mathematically, it is $\widehat{\gamma}(s, \pi) = S_0 \cup S_1 \cup \ldots$, where $S_0 = \{s_0\}$ and $S_i = \{\gamma(s, a) \mid s \in S_{i-1}\}$ for all $i > 0$.
- The *reachability graph* for s and π shows how the states in $\widehat{\gamma}(s, \pi)$ can be reached from s. Mathematically, it is a directed graph $\mathit{Graph}(s, \pi) = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \widehat{\gamma}(s, \pi)$ and $\mathcal{E} = \{(s', s'') \mid s' \in \mathcal{V}, s'' \in \gamma(s', \pi(s'))\})$.
- $\mathit{leaves}(s, \pi) = \widehat{\gamma}(s, \pi) \setminus \mathit{Domain}(\pi)$ is the set of all states that have no successors in $\mathit{Graph}(s, \pi)$.

Note that π is a *partial* function, possibly undefined in $S \setminus S'$, even for states that may have applicable actions.

An MDP problem for the domain Σ can be expressed as a triple (Σ, s_0, S_g), where $s_0 \in S \setminus S_g$ is the initial state and $S_g \subseteq S$ is a set of goal states. The problem is to reach a state in S_g starting from s_0, while possibly optimizing a cost criteria.

Definition 8.3. A *solution* to an MDP problem (Σ, s_0, S_g) is a policy $\pi : S' \rightarrow A$ such that $s_0 \in S'$ and $\mathit{leaves}(s_0, \pi) \cap S_g \neq \varnothing$. The solution is said to be *closed* if and only if $\forall s \in \widehat{\gamma}(s_0, \pi)$, $(s \in \mathit{Domain}(\pi)) \vee (s \in S_g) \vee \mathit{Applicable}(s) = \varnothing$. $\square$

In other words, every state reachable from s_0 by a closed solution π is either in the domain of π, is a goal, or has no applicable action. A closed policy π provides applicable actions, if there are any, to s_0 and to its all descendants reachable by π,

[2]Note the difference from Definition 2.2, in which $\gamma(s, a)$ is either a single state or undefined, and from Equation 2.2.

and have at least one path in $Graph(s_0, \pi)$ that reaches a goal state. It is defined over the entire $\widehat{\gamma}(s_0, \pi)$, except at goal states and states that have no applicable action. As usual, goals are considered to be *terminal* states requiring no further action.

Example 8.4. Here is a simple example, inspired from casino coin machines called *one-armed bandits*. This domain has three state variables x, y, and z, ranging over the set $\{v_a, v_b, v_c\}$. The domain has nine states: $\{x = v_a, y = v_a, z = v_a\} \ldots \{x = v_c, y = v_c, z = v_c\}$, which are abbreviated as $S = \{(aaa), (aab), \ldots, (ccc)\}$. There are three actions: pull left, pull right, and pull both arms simultaneously, denoted respectively Left, Right, and Both. When the values of the three state variables are distinct, then the three actions are applicable. If $x \neq y = z$, only Left is applicable. If $x = y \neq z$, only Right is applicable. If $x = z \neq y$, only Both is applicable. When the three variables have the same value no action is applicable. Here is a possible specification of Left (each outcome is prefixed by its corresponding probability):

```
Left:
pre:  $(x \neq y)$
eff:  $(\frac{1}{3})$:    $\{x \leftarrow v_a\}$
      $(\frac{1}{3})$:    $\{x \leftarrow v_b\}$
      $(\frac{1}{3})$:    $\{x \leftarrow v_c\}$
```

Similarly, when applicable, Right randomly changes z; Both randomly changes y. We assume these changes to be uniformly distributed. Figure 8.1 gives part of the state space of this domain corresponding to the problem of going from $s_0 = (abc)$ to a goal state in $S_g = \{(bbb), (ccc)\}$. Note that every action in this domain may possibly leave the state unchanged, that is, $\forall s \forall a,\ s \in \gamma(s, a)$. Note also that the state space of this domain is not fully connected: once two variables are made equal, there is no action to change them. Consequently, states (acb), (bac), (bca), (cab) and (cba) are not reachable from (abc).

A solution to the problem in Figure 8.2 is, for instance,

$$\pi(abc) = \text{Left}, \pi(bbc) = \pi(bba) = \text{Right}, \pi(cbc) = \pi(cac) = \text{Both}.$$

Here, π is defined over $Domain(\pi) = \{s_0, (bbc), (cbc), (bba), (cac)\}$, and $\widehat{\gamma}(s_0, \pi) = Domain(\pi) \cup S_g$. Figure 8.2 gives the $Graph(s_0, \pi)$ for that solution. □

Definition 8.3 is for *goal reachability problems*. In some domains, the actor does not have specific goals. It may want to perform a task ending with some termination action. Alternatively, it may want to keep acting optimally over an infinite horizon, a case referred to as *process maintenance problems*. We focus on goal reachability problems and discussed the other cases in Section 8.3.1.

8.1.2 Safe and Unsafe Policies

Let π be a closed solution to the problem (Σ, s_0, S_g). Run-Policy is a simple procedure for acting with a policy π, by performing in each state s the action given by $\pi(s)$ until reaching a goal or a state that has no applicable action.

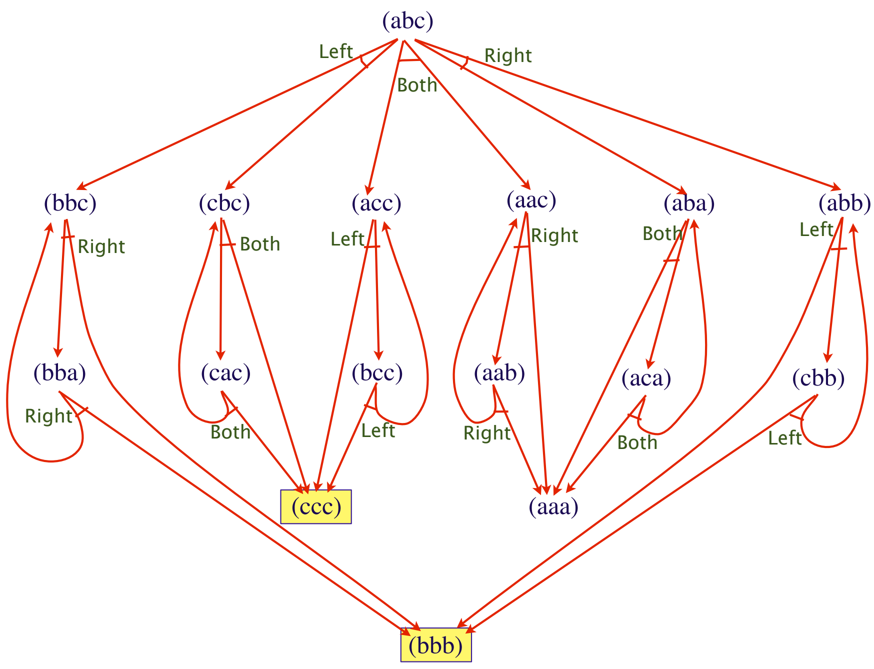


Figure 8.1. Part of the state space for the problem in Example 8.4.

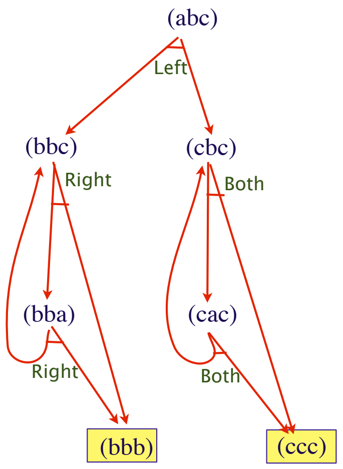


Figure 8.2. A safe solution for Example 8.4 and its *Graph*(s_0, π); self-loops in every node are implicit.

```
Run-Policy($\Sigma, s_0, S_g, \pi$)
    $s \leftarrow s_0$
    while $s \notin S_g$ and $s \in Domain(\pi)$ do
        perform action $\pi(s)$
        $s \leftarrow$ observe resulting state
```

Algorithm 8.1. Run-Policy, a simple procedure to run a closed solution policy.

Let $\sigma = \langle s_0, s_1, \ldots, s_h \rangle$ be a finite sequence of states followed by this procedure in some run of policy π that reaches a goal, that is, $s_h \in S_g$. σ is called a *history*; it is a path in $Graph(s_0, \pi)$ from s_0 to S_g. For a given π there can be an infinite number of finite histories. The cost of σ is the total sum of the cost of actions along the history σ.

$$\text{cost}(\sigma) = \sum_{i=0}^{h-1} \text{cost}(s_i, \pi(s_i), s_{i+1}), \qquad \text{for } \sigma = \langle s_0, s_1, \ldots, s_h \rangle.$$

The probability of following the history σ is $\Pr(\sigma \mid s_0, \pi) = \prod_{i=0}^{h-1} \Pr(s_{i+1}|s_i, \pi(s_i))$. Note that σ may not be a simple path: it may contain loops, that is, $s_j = s_i$ for some $j > i$. But because actions are nondeterministic, a loop does not necessarily prevent the procedure from eventually reaching a goal: the action $\pi(s_i)$ that led to an already visited state may get out of the loop when executed again at step j.

Example 8.5. For the policy in Figure 8.2, a history that reaches a goal despite visiting the same state three times is $\sigma = \langle s_0, (cbc), (cac), (cbc), (cbc), (ccc) \rangle$ □

A policy may also get trapped forever in a loop, or it may reach a nongoal leaf. Hence Run-Policy may not terminate or not reach a goal. The actor preferably seeks solutions that offer some guarantee of reaching a goal. Let $\Pr^l(S_g|s, \pi)$ be the probability of reaching a goal from a state s by following policy π for at most l steps: $\Pr^l(S_g|s, \pi) = \sum_\sigma \Pr(\sigma)$, for $\sigma \in \{\langle s, s_1, \ldots, s_h \rangle \mid s_{i+1} \in \gamma(s_i, \pi(s_i)), s_h \in S_g, h \leq l\}$. Let $\Pr(S_g|s, \pi) = \lim_{l \to \infty} \Pr^l(S_g|s, \pi)$. With this notation, it follows that:

- if π is a solution to the problem (Σ, s_0, S_g) then $\Pr(S_g|s_0, \pi) > 0$;
- a goal is reachable from a state s with policy π if and only if $\Pr(S_g|s, \pi) > 0$;
- if $s \notin Domain(\pi)$, then $\Pr(S_g \mid s, \pi)$ is 1 if $s \in S_g$, and 0 otherwise.

Definition 8.6. A solution π to an MDP problem (Σ, s_0, S_g) is *safe* if and only if $\forall s \in \widehat{\gamma}(s_0, \pi)$ there is a path from s to a goal. When π is safe $\Pr(S_g|s_0, \pi) = 1$. If $0 < \Pr(S_g|s_0, \pi) < 1$ then π is an *unsafe* solution.[3] □

With a safe policy, procedure Run-Policy(Σ, s_0, S_g, π) always (i.e., with a probability 1) reaches a goal.

The number of steps needed to reach the goal is *indefinite*, i.e., finite but not bounded *a priori*. Such a bound would require a *safe acyclic* policy (see Section 8.3.4). With

[3]Another terminology refers to *proper* and *improper* for safe and unsafe solutions.

an unsafe policy, Run-Policy may or may not terminate; if it does terminate, it may reach either a goal or a state with no applicable action.

It is useful to extend the safety concept from policies to states:

Definition 8.7. Safe, unsafe and dead end states are defined as follows:

- a state s is *safe* if and only if $\exists\pi$ such that $\Pr(S_g|s,\pi) = 1$.
- s is *unsafe* if and only if $\exists\pi$ such that $\Pr(S_g|s,\pi) > 0$ and $\forall\pi$, $\Pr(S_g|s,\pi) < 1$, or equivalently, (Σ, s, S_g) has an unsafe solution but no safe solution.
- s is a *dead end* if and only if $\forall\pi\ \Pr(S_g|s,\pi) = 0$.

An MDP problem (Σ, s_0, S_g) is safe when s_0 is safe. □

A state s is safe if and only if there exists a policy π such that for every $s' \in \widehat{\gamma}(s,\pi)$ there is a path from s' to a goal. Note that a policy π is a safe solution of (Σ, s_0, S_g) if and only if $\forall s \in \widehat{\gamma}(s_0,\pi)$, s is safe. Conversely, s is unsafe if and only if it has a dead end descendant for every policy: $\forall\pi\ \exists s' \in \widehat{\gamma}(s,\pi)\ s'$ is a dead end. If a state s is a dead end, then there is no solution to the problem (Σ, s, S_g).

A state that has no applicable action is a dead end, but so is a state from which every policy is trapped forever in a loop or leads only to other dead ends. The former are called *immediate dead ends*; the latter are *deep dead ends*.

Example 8.8. In Figure 8.1, the state (aaa) is an immediate dead end, the states (aac), (aab), (aba), and (aca) are deep dead ends, the states (bbb) and (ccc) are goals, and all of the other states are safe. Any policy starting in the safe state s_0 with either action Both or Right is unsafe because it leads to dead ends. The policy given in Figure 8.2 is safe. □

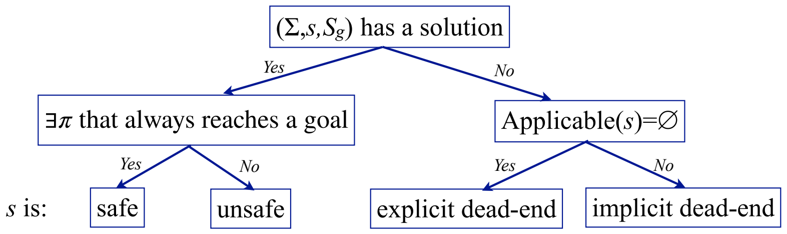


Figure 8.3. Partition of the set of states with respect to solutions.

Explicit dead ends are easy to detect: in such a state, Run-Policy(Σ, s_0, S_g, π) finds that *Applicable*$(s) = \varnothing$ and terminates unsuccessfully. Implicit dead ends create difficulties for many algorithms, including to Run-Policy that may not terminate. Figure 8.3 summarizes the four types of states with respect to goal reachability.

A domain has no dead end if and only if every state in S is safe. A domain has no reachable dead end if and only if every state reachable from s_0 by any policy is safe. These desirable cases are difficult to check in advance. A problem has a safe solution when the domain dead ends are *avoidable*: there is a π such that $\widehat{\gamma}(s_0,\pi)$ avoids dead ends. Example 8.5 illustrates a domain where dead ends are avoidable. In solving an

MDP, one will seek to avoid dead ends, searching for safe solutions. If the domain has an *unavoidable* dead end, reachable from s_0, then s_0 is unsafe. In that case, one may accept an unsafe solution whose probability of reaching a goal is maximal.

In summary, an MDP problem (Σ, s_0, S_g) can be such that:

(i) it has no dead end;
(ii) it has no reachable dead end;
(iii) it has a safe solution, i.e., its possible dead ends are avoidable; or
(iv) it has a solution, possibly unsafe.

These four cases are in decreasing order of restriction.

8.1.3 Optimal Policies

The quality of a solution π depends on how safe it is, but also how good it is with respect to an optimization criteria. The usual criteria uses the probability and cost parameters of the problem and seeks a solution with minimal expected cost.

Let us assume an MDP problem (Σ, s_0, S_g) that has a safe solution π. Let us define for π a *value function* $V^\pi : Domain(\pi) \rightarrow \mathbb{R}$ which give the expected sum of the cost of the actions obtained by following π from a state s to a goal:

$$V^\pi(s) = \mathbb{E}_\sigma\left[\sum_i \text{cost}(s_i, \pi(s_i), s_{i+1})\right], \tag{8.1}$$

where $\mathbb{E}$ is over all histories $\sigma \in \{\langle s, s_1 \ldots, s_h\rangle \mid s_{i+1} \in \gamma(s_i, \pi(s_i)), s_h \in S_g\}$.

$V^\pi(s)$ is the expected cost for running the procedure Run-Policy(Σ, s, S_g, π) from s until termination. It is the total cost of following a history σ from s to S_g, averaged over all such histories in $Graph(s, \pi)$:

$$V^\pi(s) = \sum_\sigma \Pr(\sigma)\,\text{cost}(\sigma), \tag{8.2}$$

where $\text{cost}(\sigma) = \sum_i \text{cost}(s_i, \pi(s_i), s_{i+1})$ and $\Pr(\sigma) = \prod_i \Pr(s_{i+1}|s_i, \pi(s_i))$.

Since π is assumed to be safe, the probability of reaching a goal is one; every σ is of *indefinite* length, i.e., h is finite but unbounded. Hence $V^\pi(s)$ is necessarily finite. Note for an unsafe policy the expected sum of action costs until reaching a goal is not well-defined: on a history σ on which Run-Policy(Σ, s, S_g, π) does not terminate, the sum in Equation 8.2 grows to infinity.

It is possible to prove that $V^\pi(s)$ is given by the following recursive equation (see Exercise 8.2):

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s, \pi(s))\,[\text{cost}(s, \pi(s), s') + V^\pi(s')] & \text{otherwise.} \end{cases} \tag{8.3}$$

A policy π' *dominates* a policy π if and only if $V^{\pi'}(s) \leq V^\pi(s)$ for every state for which both π and π' are defined. An *optimal policy* is a policy π^* that dominates all other policies. It has a minimal expected cost over all possible policies:

$V^*(s) = \min_\pi V^\pi(s)$. The *optimality principle* extends 8.3 to compute V^* as the fixed point of the following expression, called the *Bellman equation*:

$$V^*(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^*(s')] & \text{otherwise.} \end{cases} \tag{8.4}$$

The optimal policy is directly derived from V^*:

$$\pi^*(s) = \underset{a}{\text{argmin}} \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^*(s')] \tag{8.5}$$

In a domain that has a safe solution, one or several optimal policies π^* exist. Their value is given by the unique solution of the Bellman equation.

The value function V^π plays a critical role in solving MDP problems. It allows ranking safe policies according to their expected total cost, and to use optimization techniques for seeking a safe optimal or near optimal policy, using 8.4. V^π focuses the search heuristically on a part of the search space, possibly away from avoidable dead ends (see Chapter 9). When this is not feasible, one may accept an unsafe solution that has a high probability of reaching a goal. A trade-off between cost and probability of reaching the goal needs to be found (see Section 8.3.1).

8.2 Structured Probabilistic Representations

The modeling stage of a domain is always critical, in particular with probabilistic models. It requires good representations. The previous section used a “flat” representation, with a single state variable s that ranges over S. Such a representation requires the explicit definition of the entire state space, a requirement that is rarely feasible. Structured representations, also referred to as factored representations,[4] are exponentially more compact. They allow for the implicit definition of the ingredients of a domain through a collection of objects and parametrized state variables, as well as operators with a compact specification of probability and cost distributions, policies, and value function.

8.2.1 Probabilistic Precondition-Effect Operators

Probabilistic precondition-effect operators are a direct extension of deterministic action schemas of Section 2.3.2. Here the set $\gamma(s,a)$ and the distribution $\Pr(s'|s,a)$ are given as possible effects of an action schema, the instances of which are ground actions. Let us illustrate this representation through a few instances of a domain with increasingly more elaborate examples.

Example 8.9. Consider a simple service robot domain, called PAM$_p$, with one robot rbt and four locations $\{$pier1, pier2, exit1, exit2$\}$. At each location, there are containers of different types. The robot can move between locations; it can take a container from a location and put it in a location. The motion is deterministic, and the four locations

[4]sometime in a narrower sense, with state variables without parameters.

are pairwise adjacent. Actions take and put are constrained by the activity in the corresponding location: if it is busy, these actions fail to achieve their intended effect and do nothing. A location becomes or ceases to be busy randomly with probability p. We model this as an exogenous event, switch(l), that switches the busy attribute of location l. We assume at this stage to have a full knowledge of the state of the world. This simple domain is modeled with the following state variables:

- $\text{loc}(r) \in \{\text{pier1}, \text{pier2}, \text{exit1}, \text{exit2}\}$: location of robot r;
- $\text{ctrs}(l, \tau) \in \{0, 1, \ldots, k\}$: number of containers in location l of some type τ; we assume $\tau \in \{\text{red},\text{blue}\}$;
- $\text{load}(r) \in \{\text{red}, \text{blue}, \text{empty}\}$: type of the container on robot r if any; and
- $\text{busy}(l) \in$ Boolen.

A typical problem in PAM$_p$ is to move red containers from any of the piers to exit1 and blue ones to exit2. □

Even a domain as simple as PAM$_p$ can have a huge state space (in $O(k^l)$ for l locations), forbidding an explicit enumeration or a drawing such as Figure 8.1. An adequate specification of the actions in the previous example has to take into account their effects as well as the effects of concurrent exogenous events. Indeed, recall that nondeterminism accounts for the possible outcomes of an action a when the world is static, but also for events that may happen in the world while a is taking place, or have an impact on the effects of a. Hence, $\gamma(s, a)$ represents the set of possible states corresponding to the joint effects of a and concurrent exogenous events. The latter may however concern state variables that are not among the arguments of a. When the $|\gamma(s, a)|$ are not too large, probabilistic precondition-effect operators can be a possible representation, with en extension for free variables to handle effects of exogenous events, as illustrated next.

Example 8.10. In PAM$_p$ the deterministic effect of action move has to be combined with the random effects of events switch(l) in any of the four locations. Hence in total $|\gamma(s, \text{move})| = 2^4$. These random events are assumed to be independent and concern free variables, beyond the parameters of the action. Action move can be written as follow:

```
move($r$ : Robots; $l, m$ : Locations)
free:          $l_1, l_2, l_3, l_4$ : Locations
pre :          $\text{loc}(r) = l, l_1 \neq l_2 \neq l_3 \neq l_4$
 eff :  $p_0$ :  $\text{loc}(r) \leftarrow m$
        $p_1$ :  $\text{loc}(r) \leftarrow m, \text{busy}(l_1) \leftarrow \neg\text{busy}(l_1)$
        $p_2$ :  $\text{loc}(r) \leftarrow m, \text{busy}(l_1) \leftarrow \neg\text{busy}(l_1), \text{busy}(l_2) \leftarrow \neg\text{busy}(l_2)$
        $p_3$ :  $\text{loc}(r) \leftarrow m, \text{busy}(l_1) \leftarrow \neg\text{busy}(l_1), \text{busy}(l_2) \leftarrow \neg\text{busy}(l_2),$
               $\text{busy}(l_3) \leftarrow \neg\text{busy}(l_3)$
        $p_4$ :  $\text{loc}(r) \leftarrow m, \text{busy}(l_1) \leftarrow \neg\text{busy}(l_1), \text{busy}(l_2) \leftarrow \neg\text{busy}(l_2),$
               $\text{busy}(l_3) \leftarrow \neg\text{busy}(l_3), \text{busy}(l_4) \leftarrow \neg\text{busy}(l_4)$
```

l_1 to l_4 are free variables for the effects of random events; p_i is the probability that i switch events occur, for $i = 0$ to 4, that is, $p_0 = (1 - p)^4, p_1 = p \times (1 -$

$p)^3$, $p_2 = p^2 \times (1-p)^2$, $p_3 = p^3 \times (1-p)$, and $p_4 = p^4$. Note that there are four possible cases with probability p_1, six cases with p_2 and four cases to p_3, giving: $p_0 + 4 \times p_1 + 6 \times p_2 + 4 \times p_3 + p_4 = 1$.

The take action is similarly specified: when the robot location l is not busy and contains at least one container of the requested type c, then take may either lead to a state where l is busy with no other effect, or it may achieve its effects of a container of type τ being loaded and ctrs(l,c) being reduced by one. For each of these two cases, additional switch events may occur in any of the three other locations. This is similar for action Put (see Exercise 8.6). □

To summarize, the probabilistic actions schemas have preconditions and effects, as the deterministic schemas, but they have as many alternative sets of effects as possible outcomes. Each alternative effect field is specified with a probability value, which can be a function of the operator's parameters.

8.2.2 Dynamic Bayesian Networks

Parameterized probabilistic precondition-effect operators can be expressive, but they require going through all the alternative joint effects of an action and possible exogenous events and computing their probability. In many cases, it is not easy to factor out a large $\gamma(s, a)$ into a few alternative effects, as illustrated earlier. This representation quickly meets its limits.

Example 8.11. PAM$_q$ is a more realistic version of the PAM$_p$ domain. It takes into account the arrival of containers of different types in one of the two piers and their departure from one of the two exit locations, but it ignores the ship unloading and truck loading operations. The arrival and departure of containers and their types are considered as exogenous events. Locations have a maximum capacity of K containers of each type, K being a constant parameter. When an exit location reaches its maximum capacity for some type then the robot cannot put additional containers of that type. When a pier is full, no arrival event of the corresponding type is possible. In addition to the move, take, and put actions and the switch event seen earlier, we now have two additional events:

- arrival(l): at each state transition, if a pier l is not full and the robot is not at l then one container may arrive at l with probability q; further, 60% of arrivals in pier1 are red containers, and 80% are blue in pier2;
- departure(l) : if the robot is not at an exit location l and there are containers there, then there is a probability q' that a container may depart from l; only red containers depart from exit1 and only blue ones depart from exit2.

A typical task for the robot in domain PAM$_q$ is to move all red containers to exit1 and all blue ones to exit2. □

With only three exogenous events as in PAM$_q$, the joint effects of action and events become complex: the size and intricacy of $\gamma(s, a)$ reaches a point where the specification of precondition-effect operators is not easy (see Exercise 8.7).

Bayesian networks is an appropriate representation for expressing conditional distributions on random variables. It offers powerful techniques for reasoning on these distributions. A Bayesian network is a DAG where nodes are random variables associated with *a priori* or conditional probability distributions. An edge between two random variables x and y expresses a conditional probability dependance of y with respect to x.

Dynamic Bayesian Networks (DBNs) extend the static representation to handle different stages in time of the same variables. They are particularly convenient in our case for expressing probabilistic state transitions from s to $\gamma(s, a)$, with a focus on the state variables relevant for the action a and the exogenous events that may take place concurrently with a. This is illustrated in the following example.

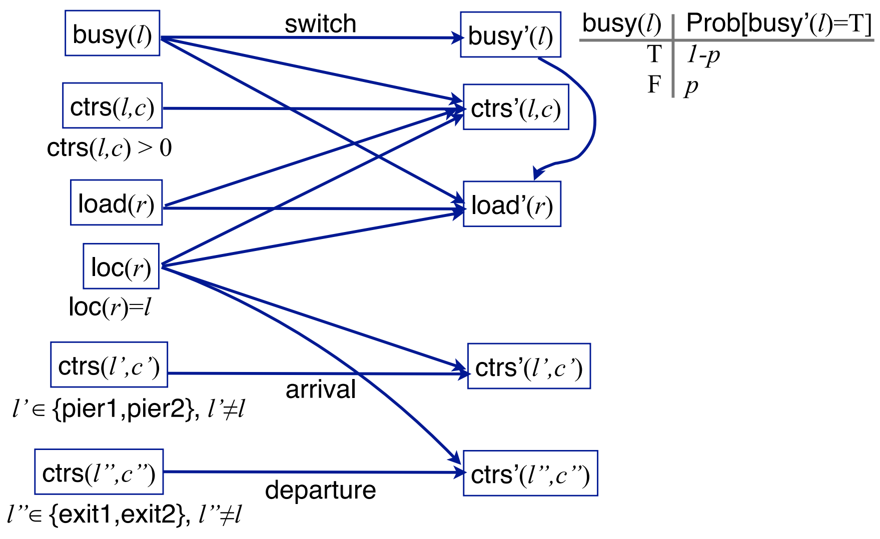


Figure 8.4. A DBN for action take in the domain PAM$_q$.

Example 8.12. Figure 8.4 represents the DBN characterizing action take in PAM$_q$ domain. It shows the state variables that condition or are affected by take and the events switch, arrival and departure. If x is a state variable of state s, we denote x' that same state variable in $s' \in \gamma(s, a)$. Here, we extend a ground DBN representation with parameterized random variables, with possible instantiation constraints. For example, busy(l) is a Boolean random variable true when location l is busy. Note that variable loc(r) conditions take but is not affected by the action and events: it appears only in the left side of the DBN.

A DBN specifies *conditional probability tables* that give the distribution over the values of a variable as a function of the values of its predecessors. Figure 8.4 illustrates such a table for the simple case of variable busy(l) that has a single predecessor. Note that p in this table varies in general with l. □

When a variable in a DBN has m ground predecessors that range over k values, the conditional probability table is of size k^m. This can quickly become a bottleneck for the specification of a DBN. Fortunately, in well-structured domains, conditional probably tables can be given in a factorized form as decision trees. These decision trees are also convenient for expressing constraints between instances of the parametrized state variables in the network.

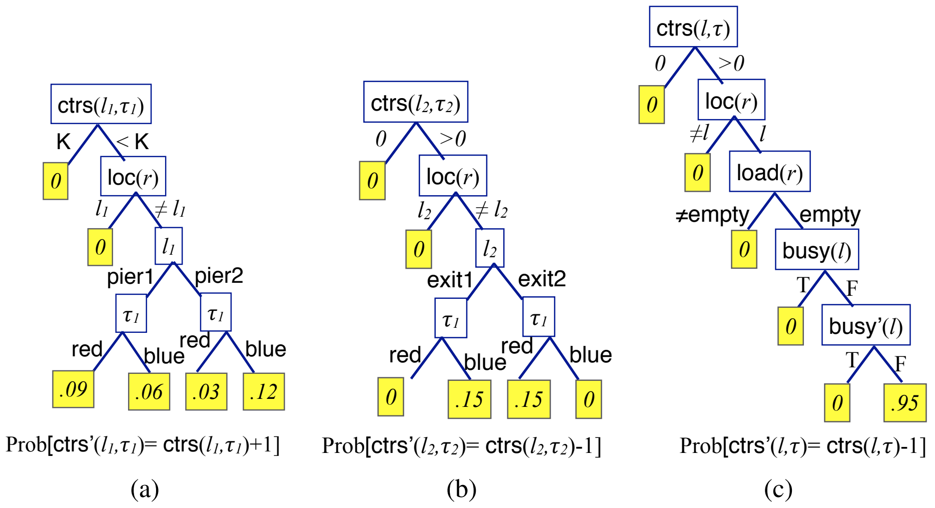


Figure 8.5. Conditional probability trees for the ctrs state variables for the action take combined with the possible events switch, arrival, and departure: (a) accounts for the arrival of a container at a pier location, (b) for a departure at an exit location, and (c) for a container being taken by the robot.

Example 8.13. Figure 8.5 gives the conditional probabilities for the ctrs variables in the DBN of Figure 8.4. The leaves of each tree give the probability that the number of containers of some type at some location increases or decreases by one container (the probability that this number remains unchanged is the complement to 1). To simplify the picture, we take $p = .05$ and $q = q' = .15$. Tree (a) accounts for the possible arrival of a container of some type at a pier location: if the location is full (ctrs(l_1, τ_1) =K) or if the robot is in that location (loc$(r) = l_1$), then no container arrival is possible, otherwise there is a probability of $.15 \times .6$ for the arrival of a red container at pier1, and so on. Similarly, tree (b) accounts for the departure of a container at an exit location. Tree (c) gives the proper effect of action take: the probability that ctrs changes is conditioned by the five ancestor variables of that node in the DBN. □

In Example 8.11, the interactions between exogenous events and actions are quite simple: events are independent and have almost no interference with the robot actions. In applications with more complex probabilistic interferences between the effects of actions and possible events, the DBN representation is especially needed. It is also convenient for the modeling of sensing actions, where sensor models must be used to relate sensed features to values of state variables.

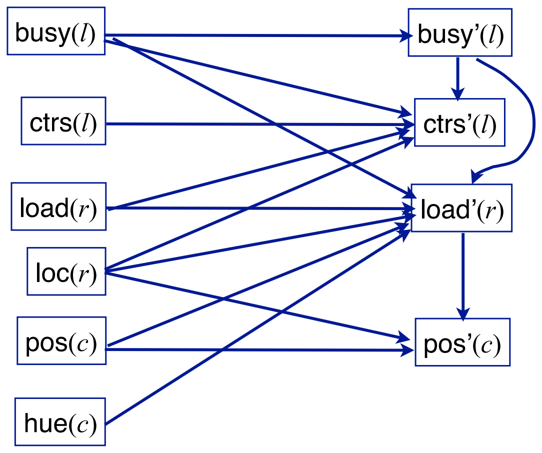
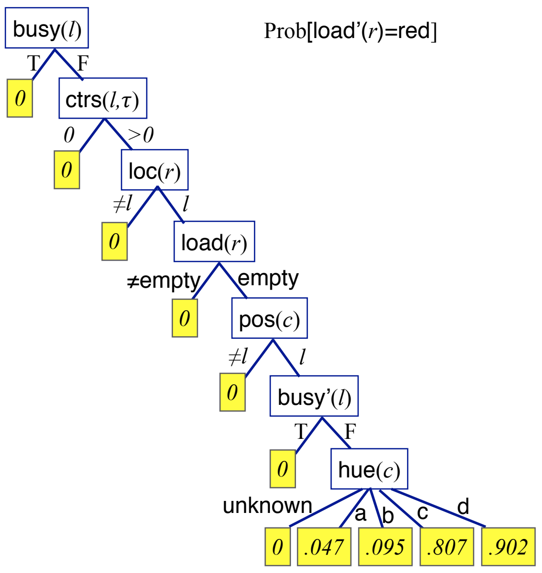


Figure 8.6. Part of the DBN for action take in domain PAM$_o$.

hue	Prob[type=red | hue]
a	*.05*
b	*.1*
c	*.85*
d	*.95*

(a)



(b)

Figure 8.7. (a) Conditional probability table for the type of a container given its observed feature; (b) conditional probability trees for load′(r)=red.

Example 8.14. Consider PAM$_o$, a variant of the previous domain where the robot does not have full knowledge of the state of the world. It still knows the exact number of containers in each location, but it does not know their types. However, it has a perceive action: when the robot is sensing a container c, perceive(c) gives the value of an observable feature, denoted hue(c), which is conditionally dependent on the container's type. To model this domain, we keep the state variables loc(r), load(r), and busy(l) as earlier; ctrs(l) is now the total number of containers in l. We further introduce the following variables:

- type(c) ∈ {red, blue}: type of container c,
- pos(c) ∈ {pier1, pier2, exit1, exit2, rbt}: location of container c, and
- hue(c) ∈{a, b, c, d, unknown}: the observed feature of c.

Action perceive(c) can be modeled as requiring the robot to be at the same location as c and hue(c) to be unknown; its effect is to change the value of hue(c) to a, b, c, or d. Furthermore, the sensor model gives a conditional probability table of type(c) given hue(c) (Figure 8.7(a)). Action take(r, l, c) is now conditioned by two additional variables pos(c), which should be identical to loc(r), and hue(c) that should be not unknown. Figure 8.6 gives a DBN for that action. A conditional probability tree for Prob[load'(r)=red] is in Figure 8.7(b). It takes into account the probability of the location becoming busy (.95), as well as the probability of looking at a red container when its observed feature has some value. Prob[load'(r)=blue] is the complement to one of the numbers in the last four leaves; it is equal to zero in the other leaves where Prob[load'(r)=empty]=1. □

The previous example illustrates two important representation issues:

- An observed feature informs probabilistically about the value of a non-observable variable. A non-observable variable (type(c) in the example) is replaced by a state variable that can be observed (here hue(c)) and to which the probabilistic planning and acting models and techniques apply normally.
- The effects of a sensing action can be required (for example, the precondition that hue(c) ≠ unknown) and planned for, as with any other state transformation action.

8.2.3 Domain Decomposition and Hierarchization

The expressiveness of structured representations for probabilistic problems allows for a compact specification of a domain that has to a huge state space, often not directly tractable with available techniques for finding a solution. In addition to a compact representation, we would like to structure a domain into smaller tractable subdomains. Two related principles can be used for that: *abstraction* and *decomposition*. Let us briefly introduce some approaches.

Abstraction methods. Abstraction consists in defining a partition of S into clusters. A cluster is a subset of states that are close enough to be considered indistinguishable with respect to some characteristics, such as to be processed jointly as a single abstract state. For example, these close states may be attributed the same policy $\pi(s)$. The original problem is solved with respect to abstract states that are these clusters, the solution of which is then possibly refined within each abstract state. Abstraction is the complement of refinement.

A popular form of abstraction is based on focusing a cluster on some relevant state variables and ignoring the other variables, considered as less relevant. The conditional probability trees in Section 8.2.2 illustrate the idea: the state variables that are not part of any tree are irrelevant. Often the irrelevant variables at one stage can be important at some other stage of the problem: the abstraction is not uniform. Furthermore, one may have to resort to approximation to find enough structure in a problem: variables that affect slightly the decision-making process (i.e., $\pi(s)$) are abstracted away.

Another abstraction approach extends model minimization techniques for computing minimal models of finite-state machines.[5] One starts with an *a priori* partition of S into clusters, for example, subset of states having (approximately) the same value function V. A cluster is split when its states have different probability transitions to states in the same or other clusters. When all clusters are homogenous with respect to state transitions, then the problem consisting of these clusters, considered as abstract states, is equivalent to the original problem. The effort in model reduction is paid off by solving a smaller problem. This is particularly the case when the clusters and the value function are represented in a factored form, as state variable formulas (see Section 9.6.2).

Symbolic algorithms (as in Section 12.3) develop this idea further with the use of algebraic decision diagrams (ADD). An ADD generalizes a decision tree into a rooted acyclic graph whose nodes are state variables, branches are possible values of the corresponding variables, and leaves are sets of states. An ADD represents a function whose values label its leaves. For example, an ADD can encode the function $V(s)$ in which all the states corresponding to a leaf have the same value. Similarly, one can represent $\Pr(s'|s,a)$ and $\text{cost}(s,a)$ as ADDs. When the structure of the problem can be mapped into compressed ADDs — a condition not easily satisfied — then fast operations on ADDs allow more efficiently exploring S or on the relevant part of it for finding a solution.

Decomposition methods. The idea is to decompose the original problem into independent or loosely coupled subproblems that are solved independently. Their solutions are *recomposed* together to get the solution of the global problem. For example, serial decomposition addresses the original task as a sequence of subtasks whose solutions will be sequentially run.

The notion of *closed subsets* of states is convenient for decomposing a domain. $C \subseteq S$ is closed if there is no transition from a state in C to a state outside of C. It is a maximal closed subset if it does not have a proper subset that is also closed. If the problem is not to reach a goal, but to control a process by acting as best as possible over an infinite horizon (process-oriented problems), an optimal policy can be constructed independently for each maximal closed subset without interfering with the rest of the domain. A maximal closed subset C can be viewed as an independent subprocess. Once reached, the system stays in this subprocess forever. C can be collapsed to a single absorbing state, at which point, other closed subsets can be found.

The *kernel decomposition* method implements this idea with more flexibility. The set S is decomposed into blocks, with possible transitions between blocks through a few states for each block. These states permitting block transitions are called the kernel of the domain. Starting with some initial value function V for the kernel states,

[5]For any given finite state machine M, there is a machine M', minimal in the number of states, which is equivalent to M, i.e., which recognizes the same language.

optimal policies are computed independently for each block, allowing one to update the values of the kernel and iterate until updates are negligible.

Finally, let us mention that abstraction and decomposition are also used for computing heuristics and control knowledge to guide or focus a global search. There is a large overlap between abstraction or decomposition methods and the techniques discussed in Section 9.3.

8.3 Modeling a Probabilistic Domain

In general, the horizon for an acting problem can be (i) *bounded*, i.e., acting stops after at most a given number h_{max} of steps, (ii) *indefinite*, i.e., the horizon is finite but not *a priori* bounded, or (iii) *infinite*. We discussed so far goal reachability MDPs that have an indefinite horizon. This section considers other cases of indefinite horizon and infinite ones, as well as issues regarding and actor's objectives, criteria, and the sources of nondeterminism it needs to model.

8.3.1 Objectives and Horizon

Let us distinguish two classes of problems corresponding to different types of horizon.

Process maintenance problems. Consider, for example, a robot whose sole function is to keep an office space clean and tidy, or a system controlling traffic lights and seeking to minimize congestions, or an elevator controller and seeking to minimize expected passengers time to their destination floors. This is the class of *process maintenance problems*, which corresponds to continual tasks. Here the actor has no specific goal state; its objective is to act optimally, over possibly an infinite horizon.

A process maintenance problem can be specified as an MDP, defined as in Definition 8.2. A solution to the problem is a policy that runs "forever," that is, as long as this policy does not prescribe an emergency exit action, or does not reach a state with no applicable action. The notions of safe and unsafe states, linked to the probability of reaching a goal, are no longer relevant as defined earlier, since there are no goals. The notion of optimal policies remains essential, but with slightly different criteria:

- for a *bounded horizon* MDP: the criteria is as expressed in Equation 8.1, but the expected sum is over all histories σ of bounded length, i.e., $|\sigma| \leq h_{max}$ for an *a priori* given bound h_{max}.
- for an *infinite horizon* MDP: the criteria is the expected sum of a *discounted* cost over an infinite horizon for a given a discount factor $0 < \delta < 1$. Equation 8.1 is changed into $V^\pi(s_0) = E[\sum_{i=0}^{\infty} \delta^i \times \text{cost}(s_i, \pi(s_i))]$; similarly, Equation 8.3 becomes $V^\pi(s) = \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s, \pi(s))[\text{cost}(s, \pi(s), s') + \delta \times V^\pi(s')]$.

Discounted cost MDP are more popular than bounded horizon ones for handling process maintenance problem. The discount factor is mathematically needed for an expected sum over an infinite horizon, but it leads to numerous drawbacks (discussed later). One can argue that there is no infinite horizon in real-life problems. Moreover, it is often easier to choose a bound h_{max} (e.g.,, the cost over one year for the traffic controller, or until the next change in the street layout), than to choose a meaningful and satisfactory discount factor δ.

Goal reachability and episodic task problems. We discussed earlier goal reachability MDP problems. Episodic task problems correspond to tasks that always terminate at some point, e.g., when performing a termination action, after a finite but unbounded number of steps. These two classes of problems are have an indefinite horizon. It is possible to model an episodic task problem as a goal reachability problem, e.g., by annotating tasks (see Section 7.2) and adding to S_g specific termination states (including failure states).

Stochastic Shortest Path (SSP) problems is an important class of goal reachability and episodic task problems. An SSP is a goal reachability MDP problem that has a safe solution and, either (i) all costs are positive, or (ii) for every unsafe policy π there is an unsafe state s such that $V^{\pi}(s) = \infty$. These condition ensure finding a safe solution and hence acting with termination. An SSP has an indefinite horizon.

SSP generalize the familiar shortest-path problems in graphs to probabilistic And/Or graphs. SSP express naturally probabilistic planning and acting problems. They are also quite general in the family of MDP models. In particular, it is possible to prove that every bounded horizon MDP and every infinite horizon discounted cost MDP problem can be restated into an equivalent SSP [131].

Goal reachability problems are sometime specified with a set of possible initial states and a probability distribution over this set. This case can be handled by adding a conventional s_0 with a single applicable action leading with the same distribution to one of initial states of the problem.

In some cases, an MDP can be addressed as satisficing problem, i.e., find any safe solution. More often MDPs are taken as optimization problem for the minimal expected cost or for other criteria, discussed next.

8.3.2 Criteria

Note that satisficing approach to a goal reachability MDP can be obtained as a particular case of optimizing with unit costs: one minimizes the expected distance to the goal, which usually leads to good heuristics for finding a solution.

Maximizing rewards. Instead of costs, one might be interested in taking into account action *rewards* for reaching particular states. Rewards are simply the opposite of costs. One switches from minimization to maximization problems. SSPs with rewards also require a safe solution and either strictly positive rewards or infinite value function for unsafe state and policies.

In problems with possibly negative costs or reward (i.e, mixing bonuses and penalties), the latter condition is hard to verify. One has to check that every cycle not containing a goal has a strictly positive cost (or strictly negative reward). This is not needed in a process maintenance problems with a discount factor allowing to handle real costs or rewards that sum up over infinite terms to finite values.

Scaling and shaping costs and rewards. It is easy to show from Equation 8.4 and 8.5 that an affine transformation of the cost function does not change the optimal policy. In other words, given constants α and β, the optimal policy is the same for the two functions $\mathrm{cost}(s, a, s')$ and $\mathrm{cost}'(s, a, s') = \alpha\, \mathrm{cost}(s, a, s') + \beta$.

A less immediate but more useful transformation is given by a property called the *shaping theorem*. Let $h : S \to \mathbb{R}$ be any function from the states to the reals. A cost shaping is a transformation of the cost function with h given by:

$$\mathrm{cost}'(s, a, s') = \mathrm{cost}(s, a, s') - h(s) + h(s').$$

The optimal policy remains unchanged with any cost shaping transformation. This can be quite beneficial for learning an optimal policy when guided towards desirable paths (see Chapter 10).

These properties of affine and shaping transformations apply to SSP with algebraic costs or rewards, and to infinite horizon MDP with a discount factor, for which shaping is expressed as $\mathrm{cost}(s, a, s') - h(s) + \delta h(s')$.

Minimizing the average cost per step. The expected average cost per step to the goal is an alternative objective function. For this criteria, Equation 8.1 takes an averaging factor of $1/h$, where h is the length of history σ. However, the criteria may prefer a longer high cost history to a short lower cost one (e.g., $c'/h' < c/h$ if $c' = 2c$ and $h' = 3h$). Of course this cannot happen if each unit cost is not lower than 1.

Maximizing the probability of reaching a goal. In many applications, one is more concerned about the probability of reaching a goal than about the expected cost of a policy. This is particularly the case when s_0 is unsafe. With this criterion, called MAX-PROB, one does not need to assume the existence of a safe policy, since one optimizes over the entire set of policies, including unsafe ones. One way of addressing this criteria is to take a reward maximization approach with the following reward function:

$$r(s, a, s') = \begin{cases} 1 & \text{if } s \in S_g, \\ 0 & \text{otherwise.} \end{cases} \tag{8.6}$$

In such a model, the expected value of a policy π is exactly the probability $\Pr(S_g|s_0, \pi)$ of reaching a goal from s_0 by following π.

Discount factor. An infinite horizon MDP requires a discount factor $0 < \delta < 1$, for the convergence of the infinite sum $V^\pi(s_0) = E[\sum_{i=0}^{\infty} \delta^i \times \mathrm{cost}(s_i, \pi(s_i))]$. It has been demonstrated that in the limit when δ approaches one, the discounted cost criteria approaches the average cost criteria over long horizon [131].

Discount factors seem an easy fix, including for problems with algebraic costs or rewards. But they have many disadvantages:

- Solutions to a discounted MDP are very sensitive to a chosen value of δ. In robotics, for example, it has been noted that values lead to unstable control.[6]

[6]According to the survey [622]"discounted formulations are frequently inadmissible in robot control".

- The literature often refers to financial applications, interest rates or amortization rates. Beyond finance, this does not give a convincing rational for discounts.
- Psychological arguments pointing that one is usually more sensitive to immediate than to long term rewards or costs may be true. But this can be seen more as a weakness, with possibly damaging effects, than a desirable feature to be taken as a model.
- Since in practice there is no infinite horizon, it is not obvious to justify and pickup a value for δ relevant to a given application. Discounting the future can in many cases be myopic and reflect a greedy bias.
- Introducing a discount just to handle unsafe policies and states with algebraic costs or rewards can be misleading towards such unsafe solutions.

In summary, discounts are seldom grounded in a practical meaning, they depreciate the future, lead to sensitive and possibly instable solutions and are not strictly needed. Throughout this book we avoid using discounted criteria.

8.3.3 Multiple Objectives

One might want to address multiple objectives and seek a compromise, e.g., between goal satisfaction probability, energy consumption, and time to reach the goal. A number of models aim at finding policies that provide an acceptable tradeoff between multiple objectives.

The simplest is the *constrained MDP* (C-MDP) , which allows optimizing a primary objective whilst keeping secondary objectives within given bounds. A C-MDP takes the form of an MDP with a vector of cost functions $\vec{cost} = [\text{cost}_0, \text{cost}_1, \ldots, \text{cost}_k]$ and a vector of upper bounds $\vec{u} = [u_1, \ldots, u_k]$. A solution to a C-MDP is a policy π minimizing the expected primary cost $V^{\pi}_{\text{cost}_0}(s_0)$ (as in a regular MDP); in addition π complies with the constraint that each expected secondary cost lies below its respective bound: $V^{\pi}_{\text{cost}_i}(s_0) \leq u_i \quad \forall i \in \{1, \ldots, k\}$.

Importantly, optimal policies for C-MDPs may be stochastic, i.e. they may map a state to a probability distribution over actions, as illustrated next.

Example 8.15. Consider an MDP with just an initial state and a goal state, and two actions leading from s_0 to the goal: go-slow, which costs 2 units of fuel and 10 units of time, and go-fast, which costs 10 units of fuel and 2 units of time. Fuel is the primary cost, and one must reach the goal with at most 6 units of time on average. The only deterministic policy satisfying the constraint is to apply go-fast, at a cost of 10 units of fuel. The optimal stochastic policy however, applies go-fast and go-slow 50% of the time each. It satisfies the constraint but costs only 6 units of fuel on average. □

Algorithms for solving C-MDPs can incorporate goal satisfaction probability similarly as a cost function – either by maximizing it if it is the primary objective, or by ensuring it exceeds a lower bound if it is a secondary objective. C-MDP capture the bi-level optimization problem of finding a policy whose expected cost is minimal, among those having the maximum probability of reaching the goal. In some applications, a low-cost solution with an acceptable probability of reaching the goal is preferred to a high probability policy with a significantly higher cost. C-MDPs support this since they allow optimizing over all policies above a given goal-probability threshold. However, other, more complex models are specifically designed to support the exploration of trade-offs between these often conflicting objectives.

This is the case of *multi-objective MDPs* (MO-MDPs), which are a general MDP model handling the optimisation of multiple objectives, represented by a vector of k cost functions. MO-MDPs are useful when the best way to combine these objectives into a single objective is unknown at planning time, when such a combination is computationally too complex to handle, or when the aim of the decision process is to help a user elicitate what an acceptable combination might be. An MO-MDP value function becomes a vector $\vec{V}^{\pi}$ whose components represent the expectations of the respective cost functions. A policy π dominates another policy π' iff $\vec{V}_i^{\pi}(s_0) \leq \vec{V}_i^{\pi'}(s_0)$ for $1 \leq i \leq k$ and $\vec{V}_j^{\pi}(s_0) < \vec{V}_j^{\pi'}(s_0)$ for at least one $j \in \{1, \ldots, k\}$. A solution to an MO-MDP is a Pareto coverage set of non-dominated stochastic policies, i.e. the largest set of policies such that no one in the set is dominated by another policy in the set. In practice since the Pareto coverage set is infinite, solution algorithms compute the extreme points of its convex hull, which consist of deterministic policies. The Pareto set of non-dominated stochastic policy is then implicitly given by the points on the surface of the polyhedron defined by the convex hull.

8.3.4 Nondeterminism

The sources of nondeterminism that one chooses to model in Σ and how they are modeled are critical issues. Except for trivial cases, no predictive model is without causes of uncertainty, but one may choose to ignore for good reasons some of them. Uncertainty can be due to sensing and information gathering actions, to exogenous events in the environment (i.e., the proper dynamic of the world and other unmodeled actors), to possible failures and other intrinsically nondeterministic actions.

Sensing and information gathering actions. These are essential sources of nondeterminism. Sensing related to a particular state variable x can be modeled as actions applicable in some states and associated with *a priori* and conditional distributions over possible values of x (for example, Prob[type|hue] in Example 8.14). Sensing actions that inform on x change the distribution of its values. Conditional distributions of state variables given observations can be obtained from probabilistic models of sensors.

Exogenous events and the proper dynamics of the environment. These are generally difficult to model deterministically as predictable events. When their possible effects interfere weakly with those of deliberate actions, events can be modeled as probability distributions over possible effects. For example, in Example 8.9, a location becomes or ceases to be busy because of some exogenous events, modeled simply as probabilistic effects of a move action. It is possible to model events as random variables whose values interfere with the outcome of an action. The DBN representation of actions can handle that directly (see Example 8.11). Conditional expressions have to be added to the probabilistic precondition-effect representation to take into account posterior probabilities given observed events.

Failures and nondeterministic actions. Intrinsically nondeterministic actions, such as throwing a dice or playing a casino machine (Example 8.4) are not generally mixed up with other actions and are dealt with specifically. Failures need certainly to be taken into account. However, the usual consideration of nominal effects versus erroneous effects of an action might not be the most relevant in practice. For example, the classical benchmark of navigating in a grid where a move action can lead to other nodes than the intended ones is often unrealistic: it does not take into account the necessary refinement of each action into lower level steps until reaching closed-loop controlled motion and localization. Further, rare events, such as component failures leading to non-modeled effects, are better handled with using specific approaches such as diagnosis and recovery.

Sparse Probabilistic Domains The degree of nondeterminism can be appreciated by the size of $|\gamma(s, a)|$ and how overlapping are the sets $\gamma(s, a)$, over applicable actions in s. In *sparse* probabilistic planning problems, $\gamma(s, a)$ remains a small set. In some cases, nondeterminism is limited to parts of the domain. Possibly, most actions are deterministic except for a few that have just two outcomes: nominal and abnormal effects. This is the case, for example, when most of the environment is known but a few areas are partially unknown, or when only sensing actions are nondeterministic, while all other actions have a unique predictable outcomes. In these cases, it is worthwhile to combine deterministic actions with probabilistic ones, and appropriate algorithm to each for finding a solution (see paragraph 9.5.2).

8.4 Acting with Probabilistic Models

8.4.1 Basic Acting Procedures

We introduced a very simple Run-Policy procedure (Algorithm 8.1) for acting using a precomputed policy π. When no policy is available and too complex to compute online, lookahead methods allow an actor to progressively elaborate its deliberation while acting, using a procedure such as MDP-Lookahead (Algorithm 8.2). This procedure calls a bounded *Lookahead* step, which searches for a partial plan rooted at s. *Lookahead* computes partially π, at least in s, and returns the corresponding action. The parameter θ sets bounds for the lookahead search. For example, θ may specify the depth of the lookahead, its maximum processing time, or use a real-time interruption mechanism corresponding to an acting deadline. The simple pseudo-code below can be extended when *Lookahead* fails by retrying with another θ.

The main difference between Run-Policy and MDP-Lookahead is the use of *Lookahead* instead of $\pi(s)$.

Working with a progressively generated policy, defined when and where it is needed, makes it possible for MDP-Lookahead to interleave planning and acting, while dealing with complexity and partial domain knowledge.

```
MDP-Lookahead($\Sigma, s_0, S_g$)
    $s \leftarrow s_0$
    while $s \notin S_g$ and $Applicable(s) \neq \emptyset$ do
1       $a \leftarrow$Lookahead $(s, \theta)$
        if $a$ = failure then return failure
        else
            perform action $a$
            $s \leftarrow$ observe resulting state
```

Algorithm 8.2. MDP-Lookahead, acting with the guidance of lookahead search.

In most cases, the step "perform action a" in Run-Policy or MDP-Lookahead procedures is not a primitive command. It requires further context dependent deliberation and refinement, which are discussed next.

8.4.2 Refining Actions

Performing an action as specified in Run-Policy or in MDP-Lookahead is seldom an atomic step. It requires refinement steps that may interfere with the current policy or the lookahead procedure. In some cases the deterministic techniques discussed in Part I and Part II can be used for performing these refinements. For example, $\pi(s)$ can be considered as an HTN task, refined in the context of s into a sequence of primitives, to be performed sequentially until reaching some $s' \in \gamma(s, \pi(s))$. A few extensions to deterministic methods for acting can be desirable when combined with probabilistic models for planning. Among these, in particular, are the following:

- When a refined action $\pi(s)$ is performed and leads back to the state s, this action may be performed again with a different refinement. After a few trials, one may also switch in s for $\pi()$ to some other applicable action $a' \neq \pi(s)$. This is equivalent to following a stochastic policy when needed.
- The refinement of $\pi(s)$ may require primitives to monitor the transition from s to a state in $\gamma(s, \pi(s))$

In general however refining probabilistic actions with deterministic techniques is not satisfactory. Because of the nondeterminism, it is not obvious to decide when the sequence in which $\pi(s)$ has been refined terminates and in which state of $\gamma(s, \pi(s))$. One needs to follow such a sequence in a conditional manner, with context dependent branches. Acting with such a class of refinement methods is very important; it will be covered in Part V.

8.5 Discussion and Bibliographic Notes

8.5.1 Foundations

Sequential decision making under uncertainty benefits from a long line of work in mathematics, starting with Andrei Markov in the 19th century, who initiated the theory of stochastic processes, now called Markov processes. The field developed extensively in the 1950s with contributions from optimal control, operations research and computer science. The *Dynamic Programming* book [109] opened the way to numerous developments, detailed into influential monographs, for example, [294, 130, 919, 131].

Many of the early developments were focused on process maintenance problems (Section 8.3.1). Goal reachability problems were also defined quite early: the analysis developed in [132], who coined the name SSP, traces back their origin to [317]. However, their development is in many aspects more recent and remains active within the artificial intelligence and automated planning communities, as illustrated with numerous articles and books, for example, [190, 764].

8.5.2 Stochastic Shortest Path Models and Constrained Models

The highly successful Markov Decision Process (MDP) class of models grew up into many extended and special cases.[7] The Stochastic Shortest Path (SSP) model is appealing for two reasons: *(i)* it is a simple and quite natural model for goal-oriented probabilistic problems, and *(ii)* it is more general than many MDP models. As demonstrated in [130] the SSP model includes as special cases the bounded horizon and the discounted MDP models. The cost shaping property is due to [847].

SSP are defined in the literature with a few variations related to how the so-called *connectivity assumption* and the *positive cycle assumption* are expressed. The first is defined either by assuming that every state is safe or that s_0 is safe. This amounts to requiring either that there is no dead end in the domain or that existing dead ends are avoidable with a safe policy starting at s_0. The second assumption is equivalent to requiring that every cycle not containing the goal has positive costs. These two assumptions should preferably be expressed as conditions that are easily testable at the specification stage of the domain. For example, demanding that every unsafe policy has infinite cost is less restrictive than constraining all costs to be positive, but it is also less easy to verify. A general approach is to allow for real costs and use algorithms able to check and avoid dead ends (see Chapter 9).

The Constrained MDP model (C-MDP) has been proposed in [34, 1102] to handle a constrained optimization objective. A special case is the bi-level optimization problem of finding a policy whose expected cost is minimal, among those having the maximum probability of reaching the goal [1103]. A generalization is given with the Multi-Objective MDPs model (MO-MDP) [1168, 959], which extends C-MDP to Pareto optimization approaches.

[7]These are, for example, C-MDP, MO-MDP, POMDP, MOMDP, CoMDP, MMDP, SIMDP, MDPIP, HMDP, HDMDP, GSSP, S^{3}P, DSSP, POSB-MDP, NEG-MDP, MAXPROB-MDP, MDP-IP, TiMDP, CPTP, Dec-MDP, Dec-SIMDP, Dec-POMDP, MPOMDP, POIPSG, and COM-MTDP.

8.5.3 Partially Observable Models

The model of *Partially Observable Markov Decision Process* (POMDP) provides an important generalization regarding the epistemic condition of an actor, that is, what it knows about the state it is in. The SSP and MDP models assume that after each state transition the actor knows which state s it has reached; it then proceeds with the action $\pi(s)$ appropriate for s. The POMDP model considers that the actor does not know its current state. It knows about the value of some observation variable $o \in O$ and two probability distributions: $\Pr(s'|s,a)$ for the transition from s to s' with a, and $\Pr(o|s',a)$ for observing o when reaching s' with a. This gives a probability distribution of possible states the actor might be in: $b(s)$, called the *actor's belief*, is the probability of being at some stage in state s. Beliefs are updated with Bayes rules. The belief of being in s' after doing a in s and observing o is:

$$b(s'|a,o) = \frac{\sum_{s\in S} \Pr(s'|s,a)\Pr(o|s',a)b(s)}{\Pr(o|b,a)}$$

where $\Pr(o|b,a) = \sum_{s,s'\in S} \Pr(s'|s,a)\Pr(o|s',a)b(s)$.

It has been demonstrated that the last observation o does not summarize the past execution, but the last belief does [59]. Hence, a POMDP problem can be addressed as an MDP problem in the belief space, which is continuous. One starts with an initial belief b_0 (initial state distribution) and seeks an optimal policy that gives for every belief point b an action $\pi(b)$, leading to a goal expressed in the belief space.

The value function is now a mapping from beliefs to real values; Bellman equation 8.4 is revised as:

$$V^*(b) = \min_a\{\sum_{s'} b(s')cost(s,a,s') + \sum_o \Pr(o|b,a)V^*(b(s'|a,o))\}$$

Several approaches generalizing MDP techniques to POMDPs have been proposed (see Section 9.6). They face significant difficulties, among which the following:

- A tremendous complexity if the belief space is discretized: each point corresponds to a subset of states. Hence, a discretized belief space is in $O(2^{|S|})$. Since $|S|$ is already exponential in the number of state variables, sophisticated algorithms and heuristics do not scale up very well. Significant modeling effort is required for decomposing a domain into small loosely coupled problems amenable to a solution. For example, a clever hierarchization technique is required for a small state space (about 600 states) to obtain a solution [897].
- A strong assumption (not always highlighted in the POMDP literature): a policy from beliefs to actions requires the action $\pi(b)$ to be applicable in *every* state s compatible with a belief b. It is not always the case that the intersection of $Applicable(s)$ over all states s compatible with b is meaningful. Sometimes, one would like to be able to choose an action that is feasible in a subset of $\pi(b)$ on the basis of states likelihood, as for example in [23].
- A termination issue: expressing the goal in the belief space is unnatural. A simple threshold on $\sum_{s\in S_g} b(s)$ may not be sufficient. This is often an argument in defense of infinite horizon discounted POMDP. But discounts in POMDP can be avoided with termination actions in episodic tasks problems [466].

- The partial observability model of POMDP is restrictive. It does not consider that part of s can be observable. An actor may distinguish between invisible and observable state variables; the latter may be visible or hidden at some point. One may act such as to observe observable variables needed for the activity, and such as to reduce the uncertainty about the states it will be in its planned course of action. Such a partial observability approach is pursued for example with the MOMDP models [858, 48], which consider that the set of states is the Cartesian product of a set of visible state variables and a set of hidden ones.
- Finally, observability issues requires a specific handling of observation actions. The set O has to be structured. At some step one does not try to observe all observable variables, but only those relevant for the current stage of the task at hand; irrelevant unknown observables are ignored. Further, it is not a single observation step; it can be a succession of observations until reducing the uncertainty to a level consistent with what's at stake. These observation actions have a cost and need to be planned for. This is for example illustrated in the HiPPo systems of [1048] for a robotics manipulation task.

We'll come back to POMDP and MOMDP in the following chapter.

8.5.4 Domain Modeling and Languages

An overview of factored MDP representations with their merits and problems is given in [170]. Their use for representing actions has been introduced in [280]. The modeling language PPDDL [1212] extends PDDL to probabilistic operators.

The Relational Dynamic Influence Diagram Language (RDDL) [979] is a compact representation integrating Dynamic Bayesian Networks and influence diagrams. Bayesian Networks are covered in the textbook [628]. RDDL allows efficiently modeling domains with exogenous events.

8.5.5 Extended MDP Models

We referred to probabilistic MDP models with timeless state transitions. Many applications require explicit time, durations, concurrency, and synchronization concepts. A simple MDP extension adds time in the state representation, for example, time as an additional state variable. In this direct extension, timeless MDP techniques can be used to handle actions with deterministic durations and goals with deadlines. However, this model cannot handle concurrent actions. The Semi-Markov Decision Process (SMDP) model [522, 368] extends this simple temporal MDP model with probabilistic integer durations. The Time-dependent MDP (TiMDP) model [173] considers distribution of continuous relative or absolute time durations. Concurrent MDPs extend the timeless MDP model to handle concurrent steps of unit duration, where each transition is a subset of actions [767]. A Generalized SMDP model combines semi-Markov models with concurrency and asynchronous events [1215]. Algorithms for these models have been proposed by several authors, notably [765, 724, 766]. It is interesting to note that SMDP provide a foundation to several reinforcement learning approaches [870, 39, 756, 350].

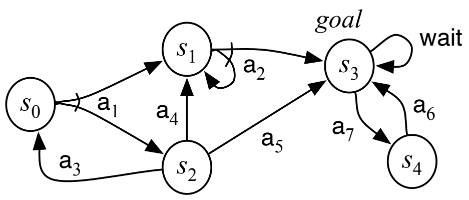
Possibilistic MDP transpose the Markov framework to cases where uncertainty is due to a lack of knowledge handled with qualitative estimates [968]. In some domains this was found to give better results than with probabilistic MDP [312]. A hybrid framework where state transitions can be modeled as either possibilistic or probabilistic, depending in the nature of actions and available information, has also been proposed [101].

Another important extension is related to *continuous* and *hybrid* state space and action space. The hybrid state space combines discrete and continuous state variables. The latter have been addressed with severable discretization techniques such as adaptive approximation [819], piecewise constant or linear approximation [345], and parametric function approximation [728, 662]. Linear Programming approaches for hybrid state spaces have been proposed by several authors, for example, [453]. Heuristic search techniques have been extended to hybrid cases, for example, the HAO* algorithm [787].

Finally, there are several extensions of the stationary and deterministic policy models. A *stochastic* policy maps states into probability distributions over actions. A *non-stationary* policy evolve with time, that is, it is a mapping of state and time into either actions when it is deterministic, or into probability distributions over actions when the policy is both stochastic and non-stationary. In some cases, such as in finite horizon problems, a non-stationary policy can be better than a stationary one, for example, $\pi(s)$ is not the same action when visiting s the first time then on the n^{th} visit. However, extending the state representation (with variables representing the context) is often easier than handling general non-stationary stochastic models, for which fewer algorithms and computational results are known (for example, [987]).

8.6 Exercises

8.1. In the SSP shown here, every action has cost 1 except wait, which has cost 0. For each action with more than one outcome, all outcomes are equally likely.



(a) How many different policies are there (excluding partial policies)? Explain.

(b) Write and solve a set of linear equations for the expected cost of the policy $\pi = \{(s_0, a_1), (s_1, a_2), (s_2, a_3), (s_3, \text{wait}), (s_4, a_5)\}$.

(c) Give an optimal policy π^*. Is there more than one such policy? What is $V^{\pi^*}(s)$ for each s?

8.2. Prove that the recursive Equation 8.3 follows from the definition of $V^\pi(s)$ in Equation 8.1.

8.3. Prove that a policy π^* that meets Equation 8.5 is optimal.

8.4. In the domain of Example 8.4, consider a policy π such that $\pi(s_0)$ = Both. Is π a safe policy when s_0 is either (acb), (bca) or (cba)? Is it safe when s_0 is (bac) or (cab)?

8.5. Consider the domain Σ in Example 8.4.

(a) Extend Σ with a fourth action denoted All, which is applicable only in the state (aaa) and flips randomly the three variables at once. Does the corresponding state space have dead ends? If not, run algorithm VI on this example, assuming uniform cost and probability distributions.

(b) Extend Σ by having the three state variables range over $\{1, 2, \ldots, m\}$, such that actions Left, Right, and Both are as defined initially; action All is applicable only to a state of the form (i, i, i) where i is even; it flips randomly the three variables. Assume $s_0 = (1, 2, 3)$ and goals are of the form (i, i, i) where i is odd. Run VI on this extended example and analyze its performance with respect to m.

8.6. Write the probabilistic precondition-effect operators for the take and put actions of the domain PAM_p (Example 8.10). How many ground actions are there is this domain?

8.7. For the domain in Example 8.11, analyze the interactions between the arrival, departure, and switch events with the action take and put. Compute the sets $\gamma(s, \text{take})$ and $\gamma(s, \text{put})$ for different states s.

8.8. Analyze a generalized PAM_q domain where the arrival and departure of containers can take place even in the robot location. Define conditional probability trees for the variable ctrs.

8.9. Model an MDP system controlling a single elevator. Consider state variables such as current-floor the elevator is in, pending-floors a list of floors demanded by on board passengers, demanded-floors a list of floors of waiting passengers, period characterize the time period in the day and week for the demand. An action move takes the elevator to a next pending floor; updates its state variables for the passengers getting out, those getting in with random destinations, and new random demand with distributions depending on the period. It also updates the period according to a particular distribution. Discuss the limitations of such a model.

9 Planning with Probabilistic Models

This chapter is about techniques for solving MDP problems. It presents planning algorithms that seeks optimal or near optimal solution policies for a domain. Most of the chapter is focused on indefinite horizon goal reachability domains that have positive costs and a safe solution; they may have dead ends but those are avoidable. This is a category of *stochastic shortest path* problems (see Section 8.3.1). Another category of SSP problems allows real costs but requires the value function to be infinite in any unsafe state. This assumption is difficult to grant when designing a domain, whereas modeling with positive costs is easier.

The chapter is organized into four main sections:

- planning algorithms based on dynamic programming and the optimality principle, i.e., policy iteration and value iteration and their extensions,
- heuristic search planning algorithms,
- linear programming approaches, capable of solving SSPs with constraints,
- online planning approaches with generative sampling models, mainly determinization techniques and Monte Carlo Tree Search techniques.

9.1 Dynamic Programming Algorithms

Dynamic programming as been the first and main algorithmic approach for solving MDP problems with an explicit state space in a flat representation. We assume here strictly positive costs (i.e., in $\mathbb{R}^+$) and domains without dead end.

9.1.1 Optimality Principle

Recall from Section 8.1.3 that the value function V^π is the expected sum of the cost of the actions obtained by following a safe policy π from a state s to a goal. Since all costs are in $\mathbb{R}^+$, $V^\pi : \mathit{Domain}(\pi) \longrightarrow \mathbb{R}^+$. We assume a domain without dead ends, hence a safe policy exists. V^π is given by Equation 8.3:

$$V^\pi(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s,\pi(s))[\mathrm{cost}(s,\pi(s),s') + V^\pi(s')] & \text{otherwise.} \end{cases}$$

A policy π' *dominates* a policy π if and only if $V^{\pi'}(s) \leq V^\pi(s)$ for every state for which both π and π' are defined. An *optimal policy* π^* dominates all other policies. It has a minimal expected cost over all possible policies: $V^*(s) = \min_\pi V^\pi(s)$. The Bellman equation 8.4 gives recursively V^* as:

$$V^*(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\mathrm{cost}(s,a,s') + V^*(s')] & \text{otherwise.} \end{cases}$$

Free pre-publication, for personal use only. To be published by Cambridge University Press.

The optimal policy π^* is derived from V^*:

$$\pi^*(s) = \operatorname*{argmin}_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^*(s')]. \tag{9.1}$$

For an arbitrary safe solution π, and V^π as in Equation 8.3, let :

$$Q^\pi(s,a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^\pi(s')]. \tag{9.2}$$

$Q^\pi(s,a)$ is called the *cost-to-go*. It is the weighted sum of the immediate cost of a in s plus the following expected cost of the successors in $\gamma(s,a)$, as estimated by V^π. Note the relation of V^π to Q^π:

$$V^\pi(s) = Q^\pi(s, \pi(s)).$$

Given a policy π, we can compute the corresponding V^π, then from V^π we can define another policy π', called the *greedy policy* for V^π, which chooses in each state the action that minimizes the cost-to-go, as estimated by V^π:

$$\begin{aligned}\pi'(s) &= \operatorname*{argmin}_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^\pi(s')] \\ &= \operatorname*{argmin}_a Q^\pi(s,a) \end{aligned} \tag{9.3}$$

In case of ties in the preceding minimum relation, we assume that the greedy policy π' keeps the value of π, that is, if $\min_a Q^\pi(s,a) = V^\pi(s)$ then $\pi'(s) = \pi(s)$.

Proposition 9.1. *When π is a safe solution, then policy π' from Equation 9.3 is safe and dominates π, that is $\forall s \; V^{\pi'}(s) \leq V^\pi(s)$. Further, if π is not optimal, then there is at least one state s for which $V^{\pi'}(s) < V^\pi(s)$.*

Starting with an initial safe policy, we can repeatedly apply Proposition 9.1 to keep improving from one policy to the next. This process converges because there is a finite number of distinct policies and each iteration brings a strict improvement in at least one state, unless already optimal. This is implemented in the Policy Iteration algorithm, detailed in the next section.

Finally, note that the Bellman equation for V (Equation 8.4) can also be expressed for Q:

$$Q^*(s,a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + \min_{a'} Q^*(s',a')] \tag{9.4}$$

from which the optimal policy is simply $\pi^*(s) = \operatorname{argmin}_a Q^*(s,a)$.

9.1.2 Policy Iteration

Policy Iteration (Algorithm 9.1), starts with an initial safe policy π_0, for example, $\pi_0(s) = \operatorname{argmin}_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)\,\text{cost}(s,a,s')$. It iteratively alternates over two phases:

- a *policy evaluation* stage which computes V^π for current π, and
- a *policy improvement* stage, which updates π with the greedy policy for the newly found V^π. Possible ties in the policy improvement are broken by giving preference to the current π, i.e., π changes only when strictly improving the minimized value.

Note the interaction between these two phases which permits convergence: updating π to the greedy policy for current V^π makes the value function no longer that of the changed π; updating V^π for the current policy makes π no longer greedy for V^π. The algorithm stops when reaching a fixed point on π, i.e., when π remains unchanged over an iteration.

```
Policy Iteration($\Sigma, S_g, \pi_0$)
    $\pi \leftarrow \pi_0$
    until reaching a fixed point on $\pi$ do
1       foreach $s \in S$ do                                   // policy evaluation
            compute $V^\pi(s)$
2       foreach $s \in S \setminus S_g$ do                     // policy improvement
            $\pi(s) \leftarrow \text{argmin}_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V^\pi(s')]$
```

Algorithm 9.1. Policy Iteration algorithm.

There are three ways of evaluating V^π for current π. The direct method observes that simple *linear equations* give V^π when π is fixed. Indeed Equation 8.3, considered over the entire S, defines a system of n linear equations, where $n = |S|$, the n unknown variables being the values of $V^\pi(s)$. There is a solution to this n linear equations if and only if the current π is safe. The value function V^π for the current π can be computed using classical linear calculs methods, such as Gaussian elimination.

A second method for finding V^π consists in computing iteratively the following series of value functions:

$$V_i(s) = \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s,\pi(s))[\text{cost}(s,\pi(s),s') + V_{i-1}(s')]. \tag{9.5}$$

It can be shown that, for any initial V_0, if π is safe, then this series converges asymptotically to a fixed point equal to V^π. In practice, one stops when $\max_s |V_i(s) - V_{i-1}(s)|$ is small enough; V_i is then taken as an estimate of V^π (more about this in the next section).

A third approach estimates V^π by computing Equation 9.5 not systematically, since this may not feasible when S is too large, but on randomly sampled paths in the MDP graph. This approach fits into the *approximate policy iteration* (API) methods. It will be detailed in Section 9.5.4.

Algorithm Policy Iteration, when initialized with a safe policy, strictly improves in each iteration the current policy over the previous one, until reaching π^*. In a domain that has no dead ends, there exists a safe π_0. All successive policies are also safe and monotonically decreasing for the dominance relation order, i.e., if the successive policies defined by Policy Iteration are $\pi_0, \pi_1, \ldots, \pi_k, \ldots, \pi^*$ then $\forall s\ V^*(s) \leq \ldots \leq V^{\pi_k}(s) \leq \ldots \leq V^{\pi_1}(s) \leq V^{\pi_0}(s)$. Because there is a finite number of distinct policies, algorithm Policy Iteration with a safe π_0 converges to an optimal policy in a finite number of iterations.

The requirement that π_0 is safe is met for domains without dead ends. If there are possible dead ends, finding a safe π_0 can be is difficult. We'll discuss later heuristic search techniques that can handle dead ends.

Generalized Policy Iteration. Policy Iteration performs the policy evaluation and policy improvement stages completely, over the entire S, at each iteration. But this is not strictly necessary. Generalized policy iteration interleaves *partial* iterative updates of the V^π and *partial* greedy improvements of the current policy, possibly on single states at a time. We'll come back later to this interesting strategy, which has been less explored for planning than for reinforcement learning where it is very convenient for a progressively discovered model of a domain (see Section 10.6).

9.1.3 Value Iteration

We defined Q^π and the greedy policy π' with respect to the value function V^π. However, the same equations 9.2 and 9.3 can be applied to any value function V, not just V^π. This gives a cost-to-go $Q^V(s,a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V(s')]$ and a greedy policy $\pi(s) = \text{argmin}_a\{Q^V(s,a)\}$ with respect to V.[1] From V, a new value function can be computed with the following equation:

$$\begin{aligned} V'(s) &= \min_a Q^V(s,a) \\ &= \min_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V(s')]. \end{aligned} \tag{9.6}$$

V' is the minimum cost-to-go in s when the successors values are estimated by V.

Dynamic programming consists in applying Equation 9.6 repeatedly, using V' as an estimate for computing the cost-to-go $Q^{V'}$, then another value function $\min_a\{Q^{V'}(s,a)\}$. This is implemented in the Value Iteration Algorithm 9.2.

Value Iteration starts with an arbitrary heuristic function V_0, which estimates the expected cost of reaching a goal from s. An easily computed heuristic is, for example, $V_0(s) = 0$ when $s \in S_g$, and $V_0(s) = \min_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)\,\text{cost}(s,a,s')$ otherwise. The algorithm iterates over improvements of the current value function by performing repeated updates using Equation 9.6. An update at an iteration propagates to $V'(s)$ changes in $V(s')$ from the previous iteration for the successors $s' \in \gamma(s,a)$. This is pursued until a fixed point is reached. A fixed point is a full iteration over S where $V'(s)$ remains identical to $V(s)$ for all s. The returned solution π is the greedy policy for the final V.

[1] The greedy policy for V is sometimes denoted π^V. When nonambiguous, in the remainder of this chapter we simply denote π the greedy policy for the current V.

```
Value Iteration($\Sigma, S_g, V_0$)
    $V \leftarrow V_0$
    until reaching a fixed point do
        foreach $s \in S \setminus S_g$ do
1           $V'(s) \leftarrow \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\mathrm{cost}(s,a,s') + V(s')]\}$
        $V \leftarrow V'$
    $\pi(s) \leftarrow \mathrm{argmin}_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\mathrm{cost}(s,a,s') + V(s')]\}$
```

Algorithm 9.2. Synchronous Value Iteration algorithm. V_0 is implemented as a function, computed once in every state; V, V' and π are global lookup tables.

Algorithm 9.2 is the *synchronous* version of Value Iteration. It implements a stage-by-stage sequence of updates where the updates at an iteration are based on values of V from the previous iteration.

```
Value Iteration($\Sigma, S_g, V_0$)
    $V \leftarrow V_0$
    until reaching a fixed point do
        foreach $s \in S \setminus S_g$ do
            Bellman-Update($s$)
```

Algorithm 9.3. Value Iteration, asynchronous algorithm. Bellman-Update uses V as a global variable.

An alternative is the *asynchronous* Value Iteration (Algorithm 9.3). There, $V(s)$ stands for the current value function for s at some stage of the algorithm. It is initialized as V_0 then repeatedly updated. An update of $V(s)$ takes into account values of successors of s and may affect the ancestors of s within that same iteration over S. In asynchronous Value Iteration, local updates are performed by the Bellman-Update Algorithm 9.4.

```
Bellman-Update($s$)
    foreach $a \in Applicable(s)$ do
        $Q(s,a) \leftarrow \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\mathrm{cost}(s,a,s') + V(s')]$
    $V(s) \leftarrow \min_a Q(s,a)$
    $\pi(s) \leftarrow \mathrm{argmin}_a Q(s,a)$
```

Algorithm 9.4. The Bellman update procedure computes $V(s)$ as in Equation 9.6, and $\pi(s)$ as the greedy policy for V. Q can be implemented as a local data structure, π and V as global data structures of algorithms using this procedure.

This procedure iterates over $a \in Applicable(s)$ to compute $Q(s,a)$ then the corresponding minimum $V(s)$ and $\pi(s)$. Several algorithms in this chapter use Bellman-Update. Throughout the chapter, we assume that ties in $\mathrm{argmin}_a\{Q(s,a)\}$, if any, are broken in favor of the previous value of $\pi(s)$ and in a systematic way (for example, lexical order of action names).

At any point of Value Iteration, either synchronous or asynchronous, an update of a state makes its ancestors no longer meeting the equation $V(s) = \min_a \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V(s')]$. A change in $V(s')$, for any successor s' of s (including when s is its own successor), requires an update of s. This is pursued until a fixed point is reached.

The termination condition of the outer loop of Value Iteration checks that a fixed point has been reached, that is, a full iteration over S without a change in V. At the fixed point, every state s meets Equation 8.3, i.e., $\forall s \;\; V(s) = V^{\pi}(s)$ for current $\pi(s)$.

In previous section, we emphasized that because there is a finite number of policies, it make sense to stop Policy Iteration when a fixed point is reached. Here, there is an infinite number of value functions; the precise fixed point is an asymptotic limit. Hence, we refine the pseudocode of Value Iteration such as to stop when a fixed point is *approximately* reached, within some acceptable margin of error. This can be assessed by the amount of change in the value of $V(s)$ during its update in Bellman-Update. This amplitude of change is called the *residual* of a state:

Definition 9.2. The *residual* of a state s with respect to V is $residual(s) = |V(s) - \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V(s')]\}|$. The global *residual* over the entire state space S is $residual = \max_{s \in S}\{residual(s)\}$. □

At each iteration of Value Iteration, $residual(s)$ is computed before each update with respect to the values of V at the previous iteration. The *termination condition* of Value Iteration with a margin of error set to a small parameter $\eta > 0$ is: $residual \leq \eta$. Note, however, that with such a termination condition, the value of $V(s)$ at the last iteration is not identical to $V^{\pi}(s)$ for current $\pi(s)$, as illustrated next.

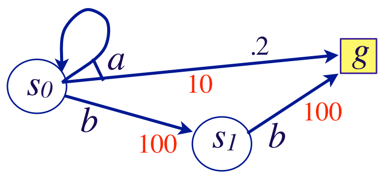


Figure 9.1. A very simple domain.

Example 9.3. Consider the very simple domain in Figure 9.1. Σ has three states, s_0, s_1, and the goal g, and two actions a and b. Action a leads in one step to g with probability p; it loops back on s_0 with probability $1 - p$. Action b is deterministic. Assume constant $\text{cost}(a) = 10, \text{cost}(b) = 100$ and $p = .2$. Σ has two solutions, denoted π_a and π_b. Their values are:

$V^{\pi_a}(s_0) = \frac{cost(a)}{p} = 50$ (directly from Equation 8.3) and

$V^{\pi_b}(s_0) = 2 \times cost(b) = 200$.

Hence $\pi^* = \pi_a$.

Let us run Value Iteration (say, the synchronous version) on this simple domain assuming $V_0(s) = 0$ in every state. After the first iteration $V_1(s_0) = 10$ and $V_1(s_1) = 100$. In the following iterations, $V_i(s_0) = 10 + .8 \times V_{i-1}(s_0)$, and $V_i(s_1)$ remains unchanged. The successive values of V in s_0 are: 18, 24.4, 29.52, 33.62, 36.89, 39.51, 41.61, 43.29, 44.63, 45.71, 46.56, and so on, which converges asymptotically to 50.

With $\eta = 10^{-4}$, Value Iteration stops after 53 iterations with solution π_a and $V(s_0) = 49.9996$. With $\eta = 10^{-3}, 10^{-2}$ and 10^{-1}, termination is reached after 43, 32, and 22 iterations, respectively. With a larger value of η, say, $\eta = 5$, termination is reached after just 5 iterations with $V(s_0) = 33.62$ (at this point: $residual(s_0) = 33.62 - 29.52 < \eta$). Note that at termination $V(s_0) \neq V^{\pi_a}(s_0)$ for the found solution π_a. We'll see next how to bound the difference $V^{\pi}(s_0) - V(s_0)$. □

Properties of Bellman updates. The iterative dynamic programming updates corresponding to Equation 9.6 have several interesting properties, which are conveniently stated with the following notation. Let (BV) be a value function corresponding to a Bellman update of V over S, that is, $\forall s\ (BV)(s) = \min_a\{Q^V(s,a)\}$. Successive updates are denoted as: $(B^kV) = (B(B^{k-1}V))$, with $(B^0V) = V$.

Proposition 9.4. *For any two value functions V_1 and V_2 such that $\forall s\ V_1(s) \leq V_2(s)$, we have: $\forall s\ (B^kV_1)(s) \leq (B^kV_2)(s)$ for $k = 1, 2, \ldots$.*

In particular, if a function V_0 is such that $V_0(s) \leq (BV_0)(s)$, then a series of Bellman updates is *monotonically nondecreasing*, in other words:
$\forall s\ V_0(s) \leq (BV_0)(s) \leq \ldots \leq (B^kV_0)(s) \leq (B^{k+1}V_0)(s) \leq \ldots$.

Proposition 9.5. *In a domain without dead end, the series of Bellman updates starting at any value function V_0 converges asymptotically to the optimal cost function V^*, that is, $\forall s\ \lim_{k\to\infty}(B^kV_0)(s) = V^*(s)$.*

Convergence and complexity of Value Iteration. For an MDP problem with positive costs and no dead ends and for any value function V_0, Value Iteration terminates. Each iteration runs in time $O(|A| \times |S|)$ (when $|\gamma(s,a)|$ is upper bounded by some constant), and the number of iterations required to reach the termination condition $residual \leq \eta$ is finite; it can be bounded under some appropriate assumptions.

Proposition 9.6. *For an MDP problem with positive costs and no dead ends,* Value Iteration *reaches termination, with residual $\leq \eta$, in a finite number of iterations.*

Regardless of the value function V_0, Value Iteration converges *asymptotically* to the optimum:

Proposition 9.7. *At termination of* Value Iteration *with residual $\leq \eta$ in an MDP problem with positive costs and no dead ends, the value V is such that $\forall s \in S$ $\lim_{\eta\to 0} V(s) = V^*(s)$.*

More precisely, it is possible to prove that at termination with V and π (the greedy policy for V), the following bound holds:

$$\forall s\ \ |V(s) - V^*(s)| \leq \eta \times \max\{\Phi^*(s), \Phi^{\pi}(s)\}, \tag{9.7}$$

where $\Phi^*(s)$ and $\Phi^\pi(s)$ are the expected number of steps to reach a goal from s by following π^* and π respectively. However, this bound is difficult to compute in the general case.

More interesting properties can be established when Value Iteration uses a heuristic function V_0 that is admissible or monotone.

Definition 9.8. V_0 is an *admissible* heuristic function if and only if $\forall s\ V_0(s) \leq V^*(s)$. V_0 is a *monotone* heuristic function if and only if $\forall s\ V_0(s) \leq \min_a\{Q(s,a)\}$. □

Proposition 9.9. *If V_0 is an admissible heuristic function, then at any iteration of* Value Iteration*, the value function V remains admissible. At termination with residual $\leq \eta$, the found value V and policy π meet the following bounds: $\forall s \quad V(s) \leq V^*(s) \leq V(s) + \eta \times \Phi^\pi(s)$ and $V(s) \leq V^\pi(s) \leq V(s) + \eta \times \Phi^\pi(s)$.*

Given π, $\Phi^\pi(s_0)$, the expected number of steps to reach a goal from s_0 following π is computed by solving the n linear equations:

$$\Phi^\pi(s) = \begin{cases} 0 & \text{if } s \in g, \\ 1 + \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s,\pi(s))\Phi^\pi(s') & \text{otherwise.} \end{cases} \tag{9.8}$$

Note the similarity between Equation 8.3 and Equation 9.8: the expected number of steps to a goal is simply V^π with unit costs. Note also that the bound $\eta \times \Phi^\pi(s_0)$ can be arbitrarily large.

Value Iteration does not guarantee a solution with an *a priori* upper-bound difference to the optimum. This difference is bounded *a posteriori*. Proposition 9.9 entails $0 \leq V^\pi(s) - V^*(s) \leq V^\pi(s) - V(s) \leq \eta \times \Phi^\pi(s)$. However, a guaranteed approximation procedure is easily defined using Value Iteration with an admissible heuristic. Algorithm VI$_\epsilon$, is a procedure to do this.

```
VI_ϵ($V_0, \epsilon$)
    $V \leftarrow V_0$; initialize $\eta > 0$ arbitrarily
    while True do
        run Value Iteration $(\Sigma, V)$
        compute $\Phi^\pi(s_0)$ for the found solution $\pi$
        if $\eta \times \Phi^\pi(s_0) \leq \epsilon$ then return
        else $\eta \leftarrow \min\{\epsilon/\Phi^\pi(s_0), \eta/2\}$
```

Algorithm 9.5. VI$_\epsilon$, a guaranteed approximation procedure for Value Iteration.

With an admissible heuristic, VI$_\epsilon$ returns a solution π within ϵ of the optimum, that is, $V^\pi(s_0) - V^*(s_0) \leq \epsilon$. It repeatedly runs Value Iteration (with V from the previous iteration) using decreasing value of η until the desired bound ϵ is reached. Notice the difference between the roles of η, the margin of error for the fixed point, and ϵ, the upper bound of the difference to the optimum.

Example 9.10. Going back to the simple domain in Example 9.3, assume we want a solution no further than $\epsilon = .1$ from the optimum. Starting with $\eta = 5$, Value Iteration finds the solution π_a after 5 iterations. Equation 9.8 for solution π_a gives $\Phi^{\pi_a}(s_0) = 5$. Value Iteration is called again with the previous V and $\eta = .02$; it stops after 23 iterations with the same solution and $V(s_0) = 49.938$. This solution is within at most .1 of π^*. Note that $V(s_0)$ is also guaranteed to be within .1 of $V^{\pi_a}(s_0)$. □

At termination of Value Iteration, $V^{\pi}(s_0)$ for the found solution π is unknown. It is bounded with: $V(s_0) \leq V^{\pi}(s_0) \leq V(s_0) + \eta \times \Phi^{\pi}(s_0)$. It is possible to compute V^{π}, as explained in Section 9.1.2, either by solving Equation 8.3 as a system of the n linear equations or by repeated updates as in Equation 9.5 until the residual is less than an accepted margin.

When the heuristic function is both admissible and monotone, then the number of iterations needed to reach termination is easily bounded. Indeed, when V_0 is monotone, then $V_0 \leq (BV_0)$ by definition, hence the remark following Proposition 9.4 applies. $V(s)$ cannot decrease throughout Bellman updates, and it remains monotone. Each iteration of increases the value of $V(s)$ for some s by at least η; it does not decrease V for any state. This entails the following bound on the number of iterations:

Proposition 9.11. *With an admissible and monotone heuristic, the number of iterations needed by Value Iteration to reach termination with residual $\leq \eta$ is bounded by* $1/\eta \sum_S [V^*(s) - V_0(s)]$.

Practical considerations. Accepting a margin of error is reasonable because the parameters of a model are always estimated with some uncertainty. There is no need to seek an exact optimal solution with respect to imprecise parameters. An approximate solution whose degree of optimality matches the accuracy of the cost and probability parameters is sufficient. An amortization trade-off takes into account how many times a suboptimal solution will be used for acting. It compares the corresponding loss in the cost of actions to the cost of further refining a suboptimal solution. For example, in a receding horizon approach in which π is used just once and recomputed at every stage, a suboptimal solution is often sufficient, whereas for a process-oriented problem the same policy is used for a long time and may require careful optimization.

When S is of a small enough size to be entirely explicited and maintained in the memory of the planning computer (typically on the order of few mega states), then Value Iteration is an easily implemented and practical algorithm. For reasonably small values of η (in the order of 10^{-3}), often Value Iteration converges in a few dozen iterations and is more efficient than Policy Iteration. Depending on the amortization trade-off, the user may not even bother to compute Φ^{π} and rely on a heuristic value of the error parameter η. There are even cases in which Value Iteration may be used online, for example, on a receding horizon schema: for $|S|$ in the order of few thousands states, the running time of Value Iteration is on the order of a few milliseconds. This may happen in small domains and in well-engineered state spaces.

Value Iteration looks as a quite efficient and scalable planning algorithm. Unfortunately, the state space in planning is exponential in the size of the input data: $|S|$ is in the order of m^k, where k is the number of ground state variables and m is the size of their range. In many practical cases k is so large (that is, a few hundred) that iterating over S is not feasible. Options in such cases are to refine the model, to decompose the problem into feasible subproblems, and to use domain configurable control knowledge to reduce sharply the branching factor of a problem, and to use focused search algorithms that explore a small part of the search space as discussed in Section 9.2 and Section 9.5.

Example 9.12. Consider a robot servicing an environment that has six locations $l_0, l_1, \ldots, l_5$, which are connected as defined by the undirected graph of Figure 9.2. Traversing an edge has a cost and a nondeterministic outcome: the tentative traversal of a temporarily busy road has no effect. For example, when in location l_0 the robot takes the action move(l_0, l_1); with a probability .5 the action brings the robot to l_1, but if the road is busy the robot has to return to l_0; in both cases the action costs 2. Edges are labelled by their traversal cost and probability of success.

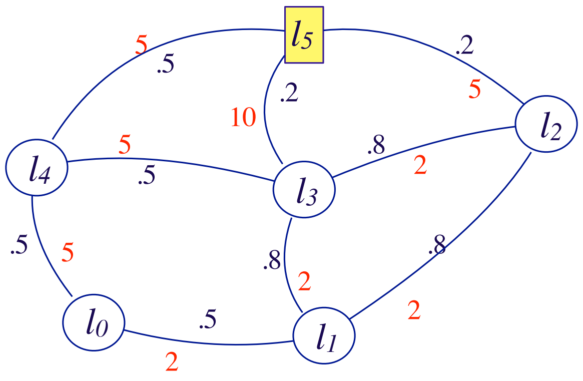


Figure 9.2. Connectivity graph of a simple environment.

In a realistic application, the robot would know (for example, from sensors in the environment) when a road is busy and for how long. Let us assume that the robot knows about a busy road only when trying to traverse it; a trial gives no information about the possible outcome of the next trial. Finding an optimal policy for traversing between two locations can be modeled as a simple MDP that has as many states as locations. A state for a location l has as many actions as outgoing edges from l; each action has two possible outcomes: reaching the adjacent location or staying in l.

Let us run Value Iteration on this simple domain for going from l_0 to l_5. With $V_0 = 0$ and $\eta = .5$, Value Iteration terminates after 12 iterations (see Figure 9.3 which gives $V(l)$ for the first three and last three iterations). It finds the following policy: $\pi(l_0)$=move(l_0, l_4), $\pi(l_4)$=move(l_4, l_5), $\pi(l_1)$=move(l_0, l_4), $\pi(l_2)$=move(l_0, l_4), $\pi(l_3)$=move(l_0, l_4). π corresponds to the path $\langle l_0, l_4, l_5 \rangle$. Its cost is $V^\pi(l_0) = 20$, which is easily computed from $V^\pi(l_4) = 5/.5$ and $V^\pi(l_0) = (5 + .5 \times V^\pi(l_4))/.5$. Note that at termination $V(l_0) = 19.87 \neq V^\pi(l_0)$. The residual after iteration 12 is $22.29 - 21.93 = .36 < \eta$.

Let us change the cost of the edge (l_0, l_4) to 10. The cost of the previous policy is now 30; it is no longer optimal. Value Iteration terminates (with the same η) after 13 iterations with a policy corresponding to the path $\langle l_0, l_1, l_3, l_5 \rangle$; its cost is 26.5. □

Figure 9.3. $V(l)$ after the first three and last three iterations of Value Iteration on the domain of Figure 9.2.

Iteration	l_0	l_1	l_2	l_3	l_4
1	2.00	2.00	2.00	3.60	5.00
2	4.00	4.00	5.28	5.92	7.50
3	6.00	7.00	7.79	8.78	8.75
10	19.52	21.86	21.16	19.76	9.99
11	19.75	22.18	21.93	19.88	10.00
12	19.87	22.34	22.29	19.94	10.00

Value Iteration versus Policy Iteration. The reader has noticed the formal similarities between Value Iteration and Policy Iteration: the two algorithms rely on repeated updates until reaching a fixed point. Their differences are worth being underlined:

- Policy Iteration approaches V^* from *above*, while Value Iteration approaches the optimum from *below*. The latter can greatly benefit from a heuristic function. Policy Iteration requires a safe initial π_0, but my also gain efficiency with a heuristic.
- Policy Iteration computes V^π for the current and final solution π, while Value Iteration relies on an approximate value of V^π for the greedy π.
- Policy Iteration reaches exactly its fixed point while a margin of error has to be set for Value Iteration, allowing for the flexibility illustrated in the procedure VI_ϵ.

Note, however, that when Policy Iteration relies on the iterative method of Equation 9.5 for computing V^π the two algorithms can be quite close.

Extensions of Value Iteration. Algorithm Value Iteration allows for several improvements and optimizations, such as ordering S according to a dynamic priority scheme, or partitioning S into acyclic components. The latter point is motivated by the fact that Value Iteration can be made to converge with just one outer loop iteration on acyclic And/Or graphs.

A variant of Value Iteration, called *Backward Value Iteration*, focuses Value Iteration by performing updates in reverse order, starting from the set of goal states, and updating only along the current greedy policy (instead of a Bellman update over all applicable actions). A symmetrical variant, *Forward Value Iteration*, performs the outer loop iteration on subsets of S, starting from s_0 and its immediate successors, then their successors, and so on.

More generally, asynchronous Value Iteration does not need to update all states at each iteration. It can be specified as follows: pick up a state s and update it. As long as the pick ups are fair, that is, no state is left indefinitely non-updated, the algorithm converges to the optimum. This opens the way to an important extension of Value Iteration for domains that have safe solutions but also dead ends. For that, two main issues need to be tackled:

- do not require termination with a fixed point for *every* state in S because this is needed only for the safe states in $\widehat{\gamma}(s_0, \pi)$ and because there may not be a fixed point for unsafe states; and

- make sure that the values $V(s)$ for unsafe states keep growing strictly such as to drive the search towards safe policies.

These issues are developed next with heuristic search algorithms.

9.2 Heuristic Search Algorithms

Heuristic search algorithms exploit the guidance of an initial value function V_0 to focus an MDP planning problem on a small part of the search space. We assume here domains with positive costs, a safe solution and possible dead ends. Before getting in the specifics of a few algorithms, let us explain their commonalities on the basis of the following search schema.

9.2.1 A General Heuristic Search Schema

The main idea of heuristic search algorithms is to explore a focused part of the search space and to perform Bellman updates within this focused part, instead of over the entire S. This explored part of the search space starts with $\{s_0\}$ and is incrementally expanded. Let the *Envelope* be the set of states that have been generated at some point by a search algorithm. The *Envelope* is partitioned into:

(i) Goal states, for which $V(s) = 0$.

(ii) *Fringe* states, non-goal states whose successors are still unknown. For a fringe state, $\pi(s)$ is not yet defined and $V(s) = V_0(s)$.

(iii) *Interior* states, whose successors are already in the *Envelope*.

Note that since this is a partition, fringe and interior states are not goals.

Expanding a fringe state s requires finding its successor states and computing $Q(s, a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s, a)[\text{cost}(s, a, s') + V(s')]$, $V(s) = \min_a\{Q(s, a)\}$, and $\pi(s) = \text{argmin}_a\{Q(s, a)\}$, the greedy policy for current V. Updating an interior state s means performing a Bellman update on s. When a descendant s' of s gets expanded or updated, $V(s')$ changes, which makes $V(s)$ no longer equal to $\min_a\{Q(s, a)\}$ and requires updating s.

Let us define the useful notions of *open* and *solved* states with respect to η, a given margin of error.

Definition 9.13. A state $s \in$ *Envelope* is *open* when s is either a fringe or an interior state such that *residual*$(s) = |V(s) - \min_a\{Q(s, a)\}| > \eta$. □

Definition 9.14. A state $s \in$ *Envelope* is *solved* when the current $\widehat{\gamma}(s, \pi)$ has no open state; i.e., s is solved when $\forall s' \in \widehat{\gamma}(s, \pi)$ either $s' \in S_g$ or *residual*$(s') \leq \eta$. □

Recall that $\widehat{\gamma}(s, \pi)$ includes s and the states in the *Envelope* reachable from s by current π. It defines *Graph*(s, π), the *current solution graph* starting from s. Throughout Section 9.2, π is the greedy policy for current V; it changes after an update. Hence $\widehat{\gamma}(s, \pi)$ and *Graph*(s, π) are defined dynamically.

Most heuristic search algorithms use the preceding notions and are based on different instantiations of a general schema called Find&Revise (Algorithm 9.6), which repeatedly performs a *Find* step followed by a *Revise* step.

```
Find&Revise($\Sigma, s_0, S_g, V_0$)
    until $s_0$ is solved do
        select an open state $s$ in $\widehat{\gamma}(s_0, \pi)$
        if $s$ is a fringe state then expand $s$
        else revise $s$
```

Algorithm 9.6. Find&Revise schema. The specifics of the *Find* and the *Revise* steps depend on the particular algorithm instantiating this schema.

The *Find* step is a traversal of the current $\widehat{\gamma}(s_0, \pi)$ for finding and choosing an open state s. This *Find* step has to be *systematic*: no state in $\widehat{\gamma}(s_0, \pi)$ should be left open forever without being chosen for revision.

The *Revise* step expands a fringe state or updates an interior state whose *residual* $> \eta$. Revising a state can change current π and hence $\widehat{\gamma}(s_0, \pi)$. At any point, either a state s is open, or s has an open descendant (whose revision will later make s open), or s is solved. In the latter case, $\widehat{\gamma}(s, \pi)$ does not change anymore.

Find&Revise iterates until s_0 is solved, that is, there is no open state in $\widehat{\gamma}(s_0, \pi)$. With an admissible heuristic function, Find&Revise converges to a solution which is asymptotically optimal with respect to η.

Proposition 9.15. *For an MDP problem with positive costs and no dead ends, and if V_0 is an admissible heuristic, then* Find&Revise *with a systematic Find step has the following properties:*

- *the algorithm terminates with a safe solution,*
- *$V(s)$ remains admissible for all states in the Envelope,*
- *the returned solution is asymptotically optimal with respect to η; its difference with V^* is bounded by: $V^*(s_0) - V(s_0) \leq \eta \times \Phi^\pi(s_0)$, where Φ^π is given by Equation 9.8, and*
- *if V_0 is admissible and monotone then the number of iterations is bounded by $1/\eta \sum_S [V^*(s) - V_0(s)]$.*

These properties are inherited from Value Iteration.

Dealing with dead ends. As discussed earlier, Dynamic Programing algorithms are limited to domains without dead ends, whereas heuristic search algorithms can overcome this limitation.

Notice first that only reachable dead ends can be of concern to an algorithm focused on the part of the state space reachable from s_0. Reachable dead ends are handled as follow:

- a deep dead end is a state s from which every action leads to an infinite loop never reaching a goal. Equation 8.1 ensures that $V(s)$ grows infinitely when s is a dead end. Indeed, $V(s)$ is the expected sum of strictly positive costs over sequences of successors of s that grow to infinite length without reaching a goal.
- an immediate dead end is a state s such that $Applicable(s) = \varnothing$ (as in Example 8.4). To make V defined in such a state, we can extend the definition by adding a third clause in Equation 8.3, stating simply: $V(s) = \infty$ if $Applicable(s) = \varnothing$. Alternatively, we can keep all the definition as introduced so far and extend the specification of a domain with a dummy action, $a_{deadend}$, applicable only in states that have no other applicable action; $a_{deadend}$ is such as $\gamma(s, a_{deadend}) = \{s\}$ and $\text{cost}(s, a_{deadend}, s) = \text{constant} > 0$. This straightforward trick brings us back to the case of deep dead ends.

Because $V(s)$ grows unbounded when s is a dead end, it is possible to show that all the properties of Find&Revise in Proposition 9.15 hold also for domains with positive costs, a safe solution in s_0, and reachable dead ends. Let us explain why this is the case:

- Since V grows infinitely for dead ends, and since an unsafe state has at least a dead end descendant for any policy and because all costs are strictly positive then $\forall s$ unsafe and $\forall \pi$, $V^{\pi}(s)$ also grows infinitely.
- With a systematic *Find*, successive Bellman updates will make at some point in two states s safe and s' unsafe: $V(s) < V(s')$. Consequently, if s is safe, the minimization $\min_a\{Q(s,a)\}$ will rule out unsafe policies.
- Finally, Find&Revise does not iterate over the entire state space but only over the current $\widehat{\gamma}(s_0, \pi)$. Since s_0 is assumed to be safe, $\widehat{\gamma}(s_0, \pi)$ will contain at some point only safe states over which the convergence to a goal is granted.

Finally, Proposition 9.15 holds also for MDP domains with algebraic cost but where $V(s)$ grows infinitely for every unsafe state. However this last assumption is not easily checked when specifying a domain.

Note that Value Iteration cannot handle dead ends as Find&Revise does, since Value Iteration iterates over the entire space S, hence cannot converge with unsafe states because there is no fixed point for dead ends (reachable or not). Heuristic search algorithms implementing a Find&Revise schema can find a near-optimal partial policy by focusing on $\widehat{\gamma}(s_0, \pi)$, which contains only safe states when s_0 is safe.

Find&Revise opens a number of design choices for the instantiation of the *Find* and the *Revise* steps and for other practical implementation issues regarding the possible memorization of the envelope and other needed data structure. Find&Revise can be instantiated in different ways, for example:

- with a best-first search, as in the algorithms AO*, LAO*, and their extensions (Section 9.2.2);
- with a depth-first and iterative deepening search, as in HDP, LDFS, and their extensions (Sections 9.2.3 and 9.2.4);

- with a stochastic simulation search, as in RTDP, LRTDP, and their extensions (Section 9.5.3).

These algorithms inherit the preceding properties of Find&Revise. They have additional characteristics, adapted to different application features. In the remainder of this chapter, we present some of them, assuming to have an MDP problem with positive cost, where s_0 is safe and V_0 is admissible.

9.2.2 Best-First Search

In deterministic planning, best-first search is illustrated with the A* algorithm for finding optimal paths in graphs. In MDP, best-first search relies on a generalization of A* for finding optimal graphs in And/Or graphs. This generalization corresponds to two algorithms: AO* and LAO*. AO* is limited to acyclic And/Or graphs, while LAO* handles cyclic search spaces. Both algorithms iterate over two steps, which will be detailed shortly:

(i) Starting at s_0, follow the current best solution graph $\widehat{\gamma}(s_0, \pi)$ to find its fringe states, and expand one of them.

(ii) Update the search space, starting at the expanded state.

The main difference between the two algorithms is in step *(ii)*. When the search space is acyclic, AO* is able to update the search space in a bottom-up stage-by-stage process focused on the current best policy. When the search space and the solution graph can be cyclic, LAO* has to combine best-first search with a Dynamic Programming update.

```
AO*($\Sigma, s_0, g, V_0$)
    $Envelope \leftarrow \{s_0\}$
    $\pi \leftarrow \varnothing; V(s_0) \leftarrow V_0(s_0)$
    while $\widehat{\gamma}(s_0, \pi)$ has fringe states do
1       traverse $\widehat{\gamma}(s_0, \pi)$ and select a fringe state $s \in \widehat{\gamma}(s_0, \pi)$
        foreach $a \in Applicable(s)$ and $s' \in \gamma(s, a)$ do
            if $s'$ is not already in $Envelope$ then
                add $s'$ to $Envelope$
                $V(s') \leftarrow V_0(s')$
2           AO-Update($s$)                    // alternatively: LAO-Update($s$)
```

Algorithm 9.7. AO*, best-first search algorithm for acyclic domains. Replacing step 2 by a call to LAO-Update(s) gives LAO*. The variables *Envelope*, π, and V are global.

Starting at s_0, each iteration of AO* (Algorithm 9.7) extracts the current best solution graph by doing a forward traversal along current π. In each branch, the traversal stops when it reaches a goal or a fringe state. The selection of which fringe state to expand is arbitrary. This choice does not change the convergence properties of the algorithm but may affect its efficiency. The expansion of a state s changes generally $V(s)$. This requires updating s and all its ancestors in the envelope

```
AO-Update($s$)
    $Z \leftarrow \{s\}$
    while $Z \neq \varnothing$ do
        select $s \in Z$ such that $Z \cap \widehat{\gamma}(s, \pi) = \{s\}$
        remove $s$ from $Z$
        Bellman-Update($s$)
        $Z \leftarrow Z \cup \{s' \in Envelope \mid s \in \gamma(s', \pi(s'))\}$
```

Algorithm 9.8. AO-Update, bottom-up update for AO*.

AO-Update (Algorithm 9.8) implements this update in a bottom-up stage-by-stage procedure, from the current state s up to s_0. The set of states that need to be updated consists of all ancestors of s from which s is reachable along current π. Note that this set is not strictly included in current $\widehat{\gamma}(s_0, \pi)$. It is generated incrementally as the set Z of predecessors of s along current π. Bellman update is applied to each state in Z whose descendants along current π are not in Z. Because the search space is acyclic, this implies that the update of a state takes into account all its known updated descendants, and has to be performed just once. The update of s redefines $\pi(s)$ and $V(s)$. The predecessors of s along π are added to Z.

A few additional steps are needed in this pseudocode for handling dead ends. The dummy action $a_{deadend}$, discussed earlier, introduces cycles; this is not what we want here. In the acyclic case, the only dead ends are immediate, that is, states not in S_g with no applicable action. This is easily detected when such a state is selected as a fringe for expansion; that state is simply labelled as a dead end. In AO-Update, for a state s that has a dead end successor in $\gamma(s, \pi(s))$, the action corresponding to $\pi(s)$ is removed from *Applicable*(s); if s has no other applicable action then s is in turn labeled a dead end, otherwise Bellman-Update(s) is performed, which redefines $\pi(s)$.

AO* on an acyclic search space terminates with a solution. When V_0 is admissible, $V(s)$ remains admissible; at termination the found solution π is optimal and $V(s_0)$ is its cost. We finally note that an efficient implementation of AO* may require a few incremental bookkeeping and simplifications. One consists in changing Z after the update of s only if $V(s)$ has changed. Another is to label solved states to avoid revisiting them. Because the space is acyclic, a state s is solved if it is either a goal or if all the successors of s in $\gamma(s, \pi(s))$ after an update are solved.

Example 9.16. Consider the domain in Figure 9.4, which has 17 states, s_0 to s_{16} and three actions a, b, and c. Connectors are labeled by the action name and cost, assumed independent of successor states; we also assume uniform probability distributions. Let us take $V_0(s) = \min_a\{\text{cost}(s, a, s')\}$ and $S_g = \{s_{12}, s_{15}, s_{16}\}$.

AO* terminates after 10 iterations, which are summarized in Figure 9.5. In the first iteration, $V(s_0) = \min\{5 + \frac{2+4}{2}, 19 + 15, 12 + \frac{5+9}{2}\} = 8$. In the second iteration, $V(s_1) = \min\{7.5, 24.5, 7\}$; the update changes $V(s_0)$, but not $\pi(s_0)$. Similarly after

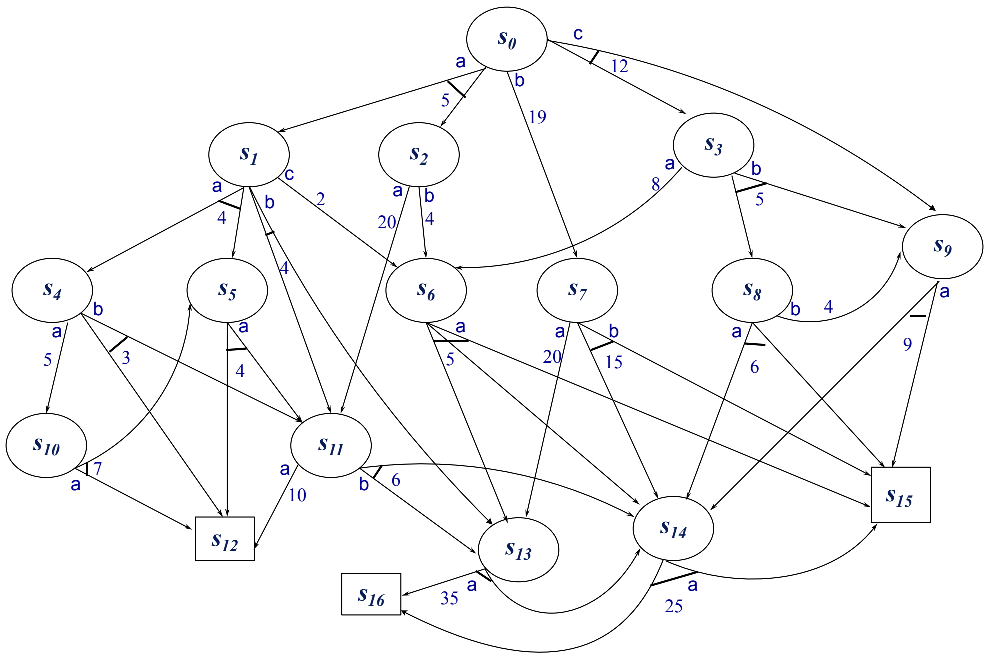


Figure 9.4. Example of an acyclic search space.

Figure 9.5. Iterations of AO* on the example of Figure 9.4: expanded state, sequence of updated states, value, and policy in s_0 after the update.

s	$V(s)$	$\pi(s)$	Updated states	$\pi(s_2)$	$\pi(s_1)$	$\pi(s_0)$	$V(s_0)$
s_0	8	a				a	8
s_1	7	c	s_0		c	a	10.5
s_2	9	b	s_0	b	c	a	13
s_6	25	a	s_2, s_1, s_0	a	a	c	19
s_3	11.5	b	s_0	a	a	a	21.75
s_4	6	b	s_1, s_0	a	a	c	22.25
s_9	21.5	a	s_3, s_0	a	a	a	22.5
s_5	7	a	s_1, s_0	a	a	a	23.5
s_{11}	10	a	s_4, s_5, s_2, s_1, s_0	b	a	a	25.75
s_{13}	47.5	a	s_6, s_2, s_1, s_0	a	a	a	26.25

s_2 is expanded. When s_6 is expanded, the updates changes $\pi(s_2), \pi(s_1)$, and $\pi(s_0)$. The latter changes again successively after s_3, s_4, and s_9 are expanded $\pi(s_0)$ = c. $\pi(s_2)$ changes after s_{11} then s_{13} are expanded. After the last iteration, the update $\pi(s_0) = \pi(s_1) = \pi(s_2) = \pi(s_5) = \pi(s_{11})$ = a and $\pi(s_4)$ = b; the corresponding solution graph has no fringe state; its cost is $V(s_0) = 26.25$.

Only 10 states in this domain are expanded: the interior states s_7, s_8, s_{10}, and s_{14} are not expanded. The algorithm performs in total 31 Bellman updates. In comparison, Value Iteration terminates after five iterations corresponding to 5×17 calls to Bellman-Update. With a more informed heuristic, the search would have been more focused (see Section 9.3 and Section 9.3). □

Let us now discuss best first search for a cyclic search space, for which updates cannot be based on a bottom-up stage-by-stage procedure. LAO* handles this general case. It corresponds to Algorithm 9.7 where step 2 is replaced by a call to LAO-Update(s). The latter (Algorithm 9.9) performs a Value Iteration-like series of repeated updates that are limited to the states on which the expansion of s may have an effect. This is the set Z of s and all its ancestors along current π. Again, Z is not limited to $\widehat{\gamma}(s_0, \pi)$.

```
LAO-Update($s$)
    $Z \leftarrow \{s\} \cup \{s' \in Envelope \mid s \in \widehat{\gamma}(s', \pi)\}$
    until termination condition do
        foreach $s \in Z$ do
            Bellman-Update($s$)
```

Algorithm 9.9. LAO-Update, q "Value Iteration-like" update for LAO*.

LAO-Update is akin to an asynchronous Value Iteration focused by current π. However, an update may change current π, which may introduce new fringe states. Consequently, the *termination condition* of LAO-Update is the following: either an update introduces new fringe states in $\widehat{\gamma}(s_0, \pi)$ or the *residual* $\leq \eta$ over all updated states.

The preceding pseudo-code terminates with a solution but no guarantee of its optimality. However, if the heuristic V_0 is admissible, then the bounds of Proposition 9.9 apply. A procedure such as VI$_\epsilon$ (Algorithm 9.5) can be used to find a solution with a guaranteed approximation.

Explicit dead ends can be handled with the dummy action $a_{deadend}$ and the management of loops. If the current π is unsafe then the updates will necessarily change that current policy, as discussed in the previous section. When there is no dead end, it is possible to implement LAO-Update using a Policy Iteration procedure, but this was not found as efficient as the Value Iteration-like procedure presented here.

LAO* is an instance of the Find&Revise schema (see Exercise 9.10). On an SSP problem with a safe solution and an admissible heuristic V_0, LAO* is guaranteed to terminate and to return a safe and asymptotically optimal solution.

The main heuristic function for driving LAO* is V_0 (see Section 9.3). Several additional heuristics have been proposed for selecting a fringe state in current $\widehat{\gamma}(s_0, \pi)$ to be expanded. Examples include choosing the fringe state whose estimated probability of being reached from s_0 is the highest, or the one with the lowest $V(s)$. These secondary heuristics do not change the efficiency of LAO* significantly. A strategy of *delayed updates* and *multiple expansions* was found to be more effective. The idea here is to expand in each iteration several fringe states in $\widehat{\gamma}(s_0, \pi)$ before calling LAO-Update on the union of their predecessors in $\widehat{\gamma}(s_0, \pi)$. Indeed, an expansion is a much simpler step than an update by LAO-Update. It is beneficial to perform updates less frequently and on more expanded solution graphs.

A variant of LAO* (Algorithm 9.10) takes this idea to the extreme. It expands all fringe states and updates all states met in a post-order traversal of current $\widehat{\gamma}(s_0, \pi)$ (the traversal marks states already visited to avoid getting into a loop). It then calls Value

```
ILAO*($\Sigma, s_0, g, V_0$)
    $Envelope \leftarrow \{s_0\}$
    while $\widehat{\gamma}(s_0, \pi)$ has fringe states do
        foreach $s$ visited in a depth-first post-order traversal of $\widehat{\gamma}(s_0, \pi)$ do
            if $s$ has not already been visited in this traversal then
                if $s$ is a fringe then expand $s$ then
                    Bellman-Update($s$)
            perform Value Iteration on $\widehat{\gamma}(s_0, \pi)$ until termination condition
```

Algorithm 9.10. ILAO*, a variant of LAO*, a best-first search algorithm for cyclic domains.

Iteration on $\widehat{\gamma}(s_0, \pi)$ with the termination condition discussed earlier. The while loop is pursued unless Value Iteration terminates with *residual* $\leq \eta$. Again, a procedure like VI$_\epsilon$ is needed to provide a guaranteed approximation.

Like AO*, LAO* can be improved by labelling solved states. This will be illustrated next with depth-first search.

9.2.3 Depth-First Search

A direct instance of the Find&Revise schema is given by the Heuristic Dynamic Programming (HDP) algorithm. HDP performs the *Find* step by a depth-first traversal of the current solution graph $\widehat{\gamma}(s_0, \pi)$ until finding an open state, which is then revised. Recall that the greedy policy for current V changes after each Bellman update. Furthermore, HDP uses this depth-first traversal for finding and labeling solved states: if s is solved, the entire graph $\widehat{\gamma}(s, \pi)$ is solved and does not need to be searched anymore.

The identification of solved states relies on the notion of *strongly connected components* of a graph. HDP uses an adapted version of Tarjan's algorithm for detecting these components (see Section A.3 and Algorithm A.4). The graph of interest here is $\widehat{\gamma}(s_0, \pi)$. Let C be a strongly connected component of this graph. Let us define a component C as being *solved* when every state $s \in C$ is solved.

Proposition 9.17. *A strongly connected component C of the current graph $\widehat{\gamma}(s_0, \pi)$ is solved if and only if C has no open state and every other component C' reachable from a state in C is solved.*

Proof. It follows from the fact that the strongly connected components of a graph define a partition of its vertices into a DAG (see Appendix A.3). If C meets the conditions of the proposition, then $\forall s \in C$, $\widehat{\gamma}(s, \pi)$ has no open state: s is solved. □

HDP (Algorithm 9.11) is indirectly recursive through a call to Solved-SCC, a slightly modified version of Tarjan's algorithm. HDP labels goal states and stops at any solved state. It updates an open state, or it calls Solved-SCC on a state s whose *residual* $\leq \eta$ to check whether this state and its descendant in the current solution

```
HDP($s$)
    if $s \in S_g$ then label $s$ solved
    if $s$ is solved then return false
1   else if $(residual(s) > \eta) \vee$ Solved-SCC($s$, false) then
        Bellman-Update($s$)
        return true
```

Algorithm 9.11. HDP, a heuristic depth-first search algorithm for SSPs.

graph are solved and to label them. Note that the disjunction (line 1) produces a recursive call only when its first clause is false. HDP and Solved-SCC returns a binary value that is true if and only if s or one of its descendants has been updated.

Solved-SCC (Algorithm 9.12) finds strongly connected components and labels them as solved if they meet the conditions of Proposition 9.17. It is very close to Tarjan's algorithm. It has a second argument that stands for a binary flag, true when s or one of its descendant has been updated. Its differences with the original algorithm are the following. In step 1 the recursion is through calls to HDP, while maintaining the *updated* flag. In step 2, the test for a strongly connected component is performed only if no update took place below s. When the conjunction holds, then s and all states below s in the depth-first traversal tree make a strongly connected component C and are not open. Further, all strongly connected components reachable from these states have already been labeled as solved. Hence, states in C are solved (see details in Section A.3).

```
Solved-SCC($s$, $updated$)
    index($s$) $\leftarrow$ low($s$) $\leftarrow i$
    $i \leftarrow i + 1$
    push($s$, $stack$)
    foreach $s' \in \gamma(s, \pi(s))$ do
        if index($s'$) is undefined then
1           $updated \leftarrow$ HDP($s'$) $\vee$ $updated$
            low($s$) $\leftarrow$ min{low($s$), low($s'$}
        else if $s'$ is in $stack$ then low($s$) $\leftarrow$ min{low($s$), low($s'$}
2   if $(\neg\, updated) \wedge$ (index($s$)=low($s$)) then
        repeat
            $s' \leftarrow$ pop($stack$)
            label $s'$ solved
        until $s' = s$
    return $updated$
```

Algorithm 9.12. Procedure for labelling strongly connected components.

HDP is repeatedly called on s_0 until it returns false, that is, until s_0 is solved.

Appropriate reinitialization of the data structures needed by Tarjan algorithm ($i \leftarrow 0$, $stack \leftarrow \varnothing$ and index undefined for states in the *Envelope*) have to be performed before each call to HDP(s_0). For the sake of simplicity, this pseudocode does not differentiate a fringe state from other open states: expansion of a fringe state (over all its successors for all applicable actions) is performed in HDP as an update step.

HDP inherits the properties of Find&Revise: with an admissible heuristic V_0, it converges asymptotically with η to the optimal solution; when V_0 is also monotone, its complexity is bounded by $1/\eta \sum_S [V^*(s) - V_0(s)]$.

9.2.4 Iterative Deepening Search

While best-first search for MDP relied on a generalization of A* to And/Or graphs, iterative deepening search relies on an extension of the IDA* algorithm.

IDA* (Iterative Deepening A*) proceeds by repeated depth-first, heuristically guided explorations of a deterministic search space. Each iteration goes deeper than the previous one and, possibly, improves the heuristic estimates. Iterations are pursued until finding an optimal path. The extension of IDA* to And/Or graphs is called LDFS; it also performs repeated depth-first traversals where each traversal defines a graph instead of a path.

```
LDFS$_a(s)$
    if $s \in S_g$ then label $s$ solved
    if $s$ is solved then return true
    $updated \leftarrow$ true
1   foreach $a \in Applicable(s)$ and while $(updated)$ do
2       if $|V(s) - \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V(s')]| \leq \eta$ then
            $updated \leftarrow$ false
3           foreach $s' \in \gamma(s,a)$ do
                $updated \leftarrow$ LDFS$_a(s') \vee updated$
    if $updated$ then Bellman-Update$(s)$
    else
        $\pi(s) \leftarrow a$
        label $s$ solved
    return $updated$
```

Algorithm 9.13. LDFS$_a$ algorithm.

We first present a simpler version of LDFS called LDFS$_a$ (Algorithm 9.13), which handles only acyclic domains. LDFS$_a$ does a recursive depth-first traversal of the current $\widehat{\gamma}(s_0, \pi)$. A traversal expands fringe states, updates open states, and labels as *solved* states that do not, and will not in the future, require updating. LDFS$_a(s_0)$ is called repeatedly until it returns s_0 as solved.

For an acyclic search space, a state s is solved when either it is a goal or when its $residual(s) \leq \eta$ and all its successors in $\gamma(s, \pi)$ are solved. This is expressed in line 2 for the current action a.

Iteration in line 1 skips actions that do not meet the preceding inequality. It proceeds recursively on successor states for an action a that meets this inequality. If these recursions returns false for all the successors in $\gamma(s, a)$, then *updated*=false at the end of the inner loop 3; iteration 1 stops and s is labeled as solved. If no action in s meets inequality in line 2 or if the recursion returns true on some descendant, then s is updated. The update is propagated back in the recursive calls through the returned value of *updated*. This leads to updating the predecessors of s, improving their $V(s)$.

Due to the test on the *updated* flag, iteration 1 does not run over all applicable actions; hence LDFS$_a$ performs *partial* expansions of fringe states. However, when a state is updated, all its applicable actions have been tried in iteration 1. Furthermore, the updates are also back-propagated partially, only within the current solution graph. Finally, states labeled as solved will not be explored in future traversals.

LDFS extends LDFS$_a$ to domains with cyclic safe solutions. This is done by handling cycles in a depth-first traversal, as seen in HDP. Cycles are tested along each depth-first traversal by checking that no state is visited twice. Recognizing solved states for cyclic solutions is performed by integrating into LDFS a book-keeping mechanism similar to the Solved-SCC procedure presented in the previous section. This integration is, however, less direct than with HDP.

Let us outline how LDFS compares to HDP. A recursion in HDP proceeds along a *single* action, which is $\pi(s)$, the current best one. LDFS examines *all* actions in *Applicable*(s) until it finds an action a that meets the condition 2 of Algorithm 9.13, and such that there is no $s' \in \gamma(s, a)$, which is updated in a recursive call. At this point, *updated*=false: iteration 1 stops. If no such action exists, then *residual*$(s) > \eta$ and both procedures LDFS and HDP perform a normal Bellman-update. Partial empirical tests show that LDFS is generally, but not systematically, faster than HDP.

LDFS is an instance of the Find&Revise schema. It inherits its convergence and complexity properties, including the bound on the number of trials when used with an admissible and monotone heuristic.

9.3 Heuristics and Search-Control Knowledge

As for all heuristic search problems, heuristic functions play a critical role in scaling up probabilistic planning algorithms. Domain-specific heuristics and control knowledge draw from *a priori* information that is not explicit in the formal representation of the domain. For example, in a stochastic navigation problem where traversal properties of the map are uncertain (for example, as in the Canadian Traveller Problem [853]), the usual Euclidian distance can provide a lower bound of the cost from a state to the goal. Domain-specific heuristics can be very informative, but it can be difficult to acquire them from domain experts, estimate their parameters, and prove their properties. Domain-specific but problem-independent efficient heuristic can be learned automatically if a simulator of the domain is available (see Chapter 10). Domain-independent heuristics do not require additional knowledge specification or a learning stage, but are often less informative. A good strategy is to combine both, relying more and more on domain-specific heuristics when they can be learned or acquired and tuned. Let us discuss here a few domain-independent heuristics and how to make use of *a priori* control knowledge.

9.3.1 Bounded-Lookahead Heuristics

A straightforward simplification of Equation 8.4 gives:

$$V_0(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)\,\text{cost}(s,a,s')\} & \text{otherwise.} \end{cases}$$

V_0 is admissible and monotone. When $|\mathit{Applicable}(s)|$ and $|\gamma(s,a)|$ are small, one may perform a Bellman update in s and use the following function V_1 instead of V_0:

$$V_1(s) = \begin{cases} 0 & \text{if } s \in S_g, \\ \min_a\{\sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}(s,a,s') + V_0(s')]\} & \text{otherwise.} \end{cases}$$

V_1 is admissible and monotone. So is the simpler variant heuristic $V_1'(s) = \min_a\{\min_{s' \in \gamma\{(s,a)}\{\text{cost}(s,a,s') + V_0(s')\}\}$ for non-goal states, because $\min_{s' \in \gamma(s,a)} V_0(s') \leq \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)V_0(s')$. This construction can straightforwardly be generalized to V_n respectively V_n' for arbitrary $n \geq 1$, unrolling the Bellman update for a lookahead of n steps. V_n and V_n' are admissible and monotone for any n. However, computational constraints limit the construction to small n, bounding the informativeness of the heuristics.

9.3.2 Determinization-Based Heuristics

A widely used relaxation for domain-independent heuristic construction is the so-called *determinization*, which transforms each probabilistic action into a few deterministic ones (as seen in Section 12.2). We can map a nondeterministic domain $\Sigma = (S, A, \gamma)$ into a deterministic one $\Sigma_d = (S, A_d, \gamma_d)$ with the following property: $\forall s \in S, a \in A, s' \in \gamma(s,a)$, $\exists a_d \in A_d$ with $s' = \gamma_d(s, a_d)$ and $\text{cost}(s, a_d) = \text{cost}(s, a, s')$. In other words, Σ_d contains a deterministic action for each nondeterministic outcome of an action in Σ. This is the *all-outcomes* determinization, as opposed to the *most-probable outcomes* determinization. In the latter, A_d contains deterministic actions only for states $s' \in \gamma(s,a)$ such that $\Pr(s'|s,a)$ is above some threshold. For SSPs in factorized representation, it is straightforward to obtain Σ_d from Σ.

Let $h^*(s)$ be the cost of an optimal path from s to a goal in the all-outcomes determinization Σ_d, with $h^*(s) = \infty$ when s is a dead end, immediate or deep. It is simple to prove that h^* is an admissible and monotone heuristic for Σ. But h^* can be computationally expensive, in particular for detecting deep dead ends. Fortunately, heuristics for Σ_d are also useful for Σ.

Proposition 9.18. *Every admissible heuristic for Σ_d is admissible for Σ.*

Proof. Let $\sigma = \langle s, s_1, \ldots, s_g \rangle$ be an optimal path in Σ_d from s to a goal; its cost is $h^*(s)$. Clearly σ is also a possible sequence of state in Σ from s to a goal with a non-null probability. No other such a history has a strictly lower cost than $h^*(s)$. Hence, $h^*(s)$ is a lower bound on $V^*(s)$, the expected optimal cost over all such histories. Let $h(s)$ be any admissible heuristics for Σ_d: $h(s) \leq h^*(s) \leq V^*(s)$. □

Hence, the techniques discussed in Section 3.2 for defining admissible heuristics, such as $h^{\max}$, are applicable in probabilistic domains. Further, informative but inadmissible heuristics in deterministic domains, such as h^{add}, have also been found informative in probabilistic domains when transposed from Σ_d to Σ.

9.3.3 Regrouped Operator-Counting Heuristics

The all-outcomes determinization relaxes probabilistic actions into deterministic actions using the very optimistic assumption that one can freely choose the outcome of every probabilistic action. This permits admissible determinization-based heuristics. However, this may also lead to harsh underestimations of the actual expected cost V^*.

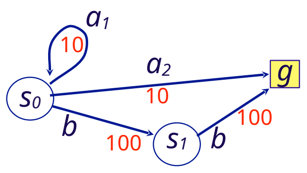


Figure 9.6. All-outcomes determinization of the simple domain from Figure 9.1.

Example 9.19. Reconsider the simple domain from Figure 9.1. Assume that the probability of a leading from s_0 to g within one step is reduced to $p = .02$, looping back to s_0 with a probability of .98. Figure 9.6 shows the all-outcomes determinization Σ_d. Σ_d treats equally all probabilistic action outcomes no matter of their likelihood. The optimal plan for s_0 in Σ_d is $\langle a_2 \rangle$. Therefore $h^*(s_0) = 10 < 200 = V^*(s_0)$. □

The *regrouped operator-counting heuristics* h^{roc} was the first family of domain-independent admissible heuristics taking into account uncertainty about the actions' outcomes. The operator-counting heuristic has its origins in deterministic models, where heuristic estimates are derived from a characterization of plans based on action-occurrence counts Count_a using linear program (LP). Formulating properties of plans as constraints over the action-occurrence counts, and choosing the LP's objective function to minimize plan cost $\sum_{a \in A} \mathsf{Count}_a \mathsf{Cost}(a)$, the optimal LP solutions yield admissible heuristics.

Let us consider two such heuristic functions $h^{\text{oc}}(s)$ and $h^{\text{roc}}(s)$. The operator-counting heuristic $h^{\text{oc}}(s)$ is given by the value of the optimal solution of the following linear program;

$$\begin{gathered} \min_{\mathsf{Count}} \sum_{a \in A} \mathsf{Count}_a \mathsf{Cost}(a) \\ \text{subject to constraints } \mathsf{Count}_a \geq 0 \text{ and } \Gamma(s). \end{gathered} \tag{9.9}$$

This formulation leaves open the choice of the *operator-counting constraints* $\Gamma(s)$. Admissibility is granted when $\Gamma(s)$ is satisfied for every plan π by the assignment:

$$\mathsf{Count}_a = \text{number of occurrences of } a \text{ in } \pi$$

for all actions a. A popular instantiation of $\Gamma(s)$ is given by action landmarks (cf. Section 3.2.3). Recall that a set of actions R is an action landmark for s if R contains at least one action from every plan. This naturally translates into the following operator-counting constraint:

$$\sum_{a \in R} \mathsf{Count}_a \geq 1 \tag{9.10}$$

h^{oc} can also admissibly combine multiple action landmarks $R_1, \ldots, R_n$ by considering in $\Gamma(s)$ the conjunction of the corresponding operator-counting constraints (9.10).

The regrouped operator-counting heuristic h^{roc} lifts the operator-counting heuristic h^{oc} to probabilistic models Σ. The core still is the linear program (9.9) derived from Σ_d, the all-outcomes determinization of Σ. In addition, h^{roc} *regroups* the determinized actions of the same probabilistic action, synchronizing their counts according to their associated outcome probabilities. To this end, h^{roc} includes for every pair of determinized actions a_d and a'_d in Σ_d of any probabilistic action a in Σ, with associated outcome probabilities p and p', the regrouping constraint:

$$\frac{1}{p}\mathsf{Count}_{a_d} = \frac{1}{p'}\mathsf{Count}_{a'_d} \tag{9.11}$$

Example 9.20. Reconsider Example 9.19 and the landmark $R = \{a_2, b\}$ for s_0 in Σ_d. The corresponding regrouped operator-counting heuristic is defined via the linear program

$$\begin{aligned}
\min_{\mathsf{Count}} \ & \mathsf{Count}_{a_1} 10 + \mathsf{Count}_{a_2} 10 + \mathsf{Count}_b 100 \\
\text{subject to } \ & \mathsf{Count}_{a_1} \geq 0 \, \mathsf{Count}_{a_2} \geq 0 \, \mathsf{Count}_b \geq 0 \\
& \mathsf{Count}_{a_2} + \mathsf{Count}_b \geq 1 \\
& \frac{50}{49}\mathsf{Count}_{a_1} = 50\mathsf{Count}_{a_2}
\end{aligned}$$

The optimal solution is $\mathsf{Count}_{a_1} = \mathsf{Count}_{a_2} = 0$ and $\mathsf{Count}_b = 1$, giving the heuristic $h^{\text{roc}}(s_0) = 100$, much higher than $h^*(s_0) = 10$ for the optimal plan of Σ_d. □

$h^{\text{roc}}(s)$ is admissible but in general not monotone. Monotonicity is violated if the constraints $\Gamma(s)$ and $\Gamma(s')$ of a state s and one of its successors s' are inconsistent, such as when using different landmark sets for s and s'. Admissibility follows from the fact that every safe policy π can be transformed into an LP solution with objective value equal to the expected cost of π, deriving the action counts Count_{a_d} of each determinized action a_d from the expected number of executions of the corresponding probabilistic action a when running π from the state s.

9.3.4 Probabilistic-Abstraction Heuristics

The idea here is to use a state abstraction function $\alpha : S \mapsto S_\alpha$, which maps several states into a single abstract state. In S_α, the distinction between states $s \neq s'$ is neglected when $\alpha(s) = \alpha(s')$. A domain Σ is mapped into a much smaller abstract domain Σ_α. The optimal expected costs V^*_α for Σ_α is taken as a heuristic estimates for Σ. This extends the abstraction heuristics of Section 3.6.5 to probabilistic domains. When Σ_α has a small size V^*_α-values can be computed for all states via methods like Value Iteration. The abstraction heuristic $h^\alpha(s) = V^*_\alpha(\alpha(s))$ is admissible and monotone.

Crucial for the efficacy of the heuristic, the abstraction function α must allow constructing Σ_α without relying on an explicit description of Σ. *Probabilistic pattern databases* are one of the most successful probabilistic abstraction heuristics. They leverage so called *syntactic projections* to construct α and Σ_α directly from the factored domain description. Given a probabilistic planning problem in state-variable representation (O, R, X, A, s_0, g), a subset of state variables $X' \subseteq X$, called a *pattern*, defines an abstraction function $\alpha_{X'}$ which projects every state to the state variables in X', i.e., $\alpha_{X'}(s) = \{x = v \mid \text{for } x = v \text{ in } s \text{ and } x \in X'\}$. The projected planning problem is (O, R, X', A', s'_0, g') where A', s'_0, and g' are obtained by discarding all appearances of the variables $X \setminus X'$ from A, s_0, and g. Given that the size of S_α scales exponentially in $|X'|$, this puts a limitation on the size of X', and therewith also on the informativeness of the corresponding projection heuristic $h^{\alpha_{X'}}$.

Pattern database heuristics compensate weaknesses of individual projections by combining a collection of patterns $C = \{X_1, \ldots, X_n\}$. A straightforward way to coalesce the estimates of the individual projection heuristics into an admissible and monotone heuristic is to take the maximum $\max_{X' \in C} h^{\alpha_{X'}}(s)$. This has the nice property that the resulting heuristic dominates each of its members. Ideally one would however want to sum up the individual estimates, which in turn dominates the maximum, but unfortunately this is not admissible in general.

Two patterns X_1 and X_2 are said to be *additive*, if there is no action with an effect on variables from both X_1 and X_2. This criterion implies that the optimal policies of the respective syntactic projections use disjoint sets of actions, which is in general sufficient to guarantee that $h^{\alpha_{X_1}}(s) + h^{\alpha_{X_2}}(s)$ is admissible. Similarly, since the two heuristics never count the same action and since they both are monotone, their sum is guaranteed to be monotone as well. The notion of additivity is extended to sets of patterns. A collection C_a is additive if all $X_i, X_j \in C_a$ are pairwise additive. Like for pairs, the additivity property is sufficient for guaranteeing that the sum $\sum_{X' \in C_a} h^{\alpha_{X'}}(s)$ is admissible and monotone.

The *canonical pattern database heuristic* h^C uses this observation to combine the projections of arbitrary collections of patterns C by identifying first all the additive subsets $C_a \subseteq C$, and taking the maximum over the respective sums:

$$h^C(s) = \max_{\text{additive } C_a \subseteq C} \sum_{X' \in C_a} h^{\alpha_{X'}}(s)$$

9.3.5 Other Search-Control Knowledge

Here, we seek to use domain-specific control knowledge in order to focus the search in a state s on a subset of applicable actions in s. *Domain-configurable* planners rely on this idea. The control knowledge can be expressed as pruning rules written in temporal logic for forward search state-space planners, or as task decomposition methods for HTN planners.

Let the focus subset be $Focus(s, \mathcal{K}) \subseteq Applicable(s)$, where $\mathcal{K}$ is the control knowledge applicable in s. Convenient approaches allow computing $\mathcal{K}$ incrementally, for example, with a function Progress such that $\mathcal{K}' \leftarrow \text{Progress}(s, a, \mathcal{K})$. In deterministic state-space planners, $\mathcal{K}$ can be a control formula; $Focus(s, \mathcal{K})$ are the applicable actions that meet this formula; Progress computes *Focus* for $\gamma(s, a)$. The planner limits its options to *Focus* and reduces its branching factor.

Two ingredients are needed to transpose these approaches to probabilistic domains: *(i)* a forward-search algorithm, and *(ii)* a representation and techniques for computing $Focus(s, \mathcal{K})$ and $\text{Progress}(s, a, \mathcal{K})$ for nondeterministic actions. The latter can be obtained from Σ_d, the determinized version of a domain. For the former, there is the forward variant of Value Iteration, and most instances of the Find&Revise schema, including best-first and depth-first, perform a forward search. Control methods can also be applied to online and anytime lookahead algorithms of Section 9.5. They can efficiently speed up a search, but they evidently reduce its convergence (e.g., with respect to safe and optimal solutions) to the actions selected in the *Focus* subset.

9.4 Linear Programming Approaches

Linear Programming (LP) is one of the oldest methods for solving MDPs. In comparison with dynamic programming, it produces exact solutions, naturally represents stochastic policies, and elegantly deals with constraints. It is capable of producing exact optimal solutions for constrained MDPs (C-MDPs) (see Section 8.3.3). It is also used as a component of heuristic search approaches for constrained stochastic shortest path problems (C-SSPs), and as a basis for deriving heuristics for SSPs.

We assume, as in Section 9.2, that all actions costs are strictly positive,[2] and that a safe policy exists from the initial state. We will briefly discuss relaxations of the latter assumption to handle unavoidable dead ends. To simplify the exposition of the linear programs, we will use cost functions $\text{cost}(s, a)$ that do not depend on the next state of the transition. In the following we abbreviate $Applicable(s)$ with $A(s)$ and write $\mathbf{I}_s$ for the function that assigns 1 to state s and zero to any other state.

9.4.1 Linear Programs for SSPs

There are two main linear programs for SSPs. The first is the Primal LP (Algorithm 9.14, in the usual format for linear programming, i.e., giving the linear criteria to be optimized and the constraints to be met). Primal LP operates in the space of value functions. It optimizes over variables V_s representing the value function at each state s, under two constraints capturing the value function definition at goal states (C1) and non-goal states (C2), respectively. The optimal solution of Primal LP is V^*.

[2] We will discuss models in which certain costs only need to be constrained rather than optimized, and those where costs are arbitrary.

Slight variants deal with rewards, infinite horizon discounted MDPs, and distributions over initial states.[3]

The Primal LP requires that there are no dead ends. If any state $s \in S$ is a dead end (even if avoidable), then the LP's objective is unbounded, hence there is no optimal solution. We could allow avoidable dead ends reachable from the initial state s_0 by changing the objective to maximize V_{s_0}. This then only guarantees that the LP solution is V^* for the states reached by some optimal policy.

```
Primal LP$(\Sigma, S_g)$
    $\max \sum_{s \in S} V_s$
        s.t. $V_s = 0$                                                       $\forall s \in S_g$   (C1)
             $V_s \leq \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\mathrm{cost}(s,a) + V_{s'}]$   $\forall s \in S, a \in A(s)$   (C2)
```

Algorithm 9.14. Primal Linear Program for SSPs

```
Dual LP$(\Sigma, s_0, S_g)$
    $\min \sum_{s \in S \setminus S_g, a \in A(s)} \mathrm{cost}(s,a) x_{s,a}$
        s.t. $x_{s,a} \geq 0$                                          $\forall s \in S \setminus S_g, a \in A(s)$   (C3)
             $out(s) = \sum_{a \in A(s)} x_{s,a}$                         $\forall s \in S \setminus S_g$   (C4)
             $in(s) = \sum_{s' \in S \setminus S_g, a \in A(s')} x_{s',a} \Pr(s|s',a)$      $\forall s \in S$   (C5)
             $out(s) - in(s) = \mathbf{I}_{s_0}(s)$                         $\forall s \in S \setminus S_g$   (C6)
             $\sum_{s \in S_g} in(s) = 1$                                                    (C7)
```

Algorithm 9.15. Dual Linear Program for SSPs

The second linear program is the Dual LP (Algorithm 9.15). It operates in the space of stochastic policies. A stochastic policy is a function $\pi : S \times A \mapsto [0,1]$ which returns a probability distribution over actions to be performed in a given state; $\pi(s,a)$ is the probability required by π of performing a in s. The Dual LP optimizes

[3] It suffices to replace (C1) with an assertion that the V_s are positive, add the discount factor and invert the direction of the inequality in (C2), and change the objective to minimize the sum of the state values weighted by their initial probabilities.

over variables representing the *occupation measures* of the policy. The occupation measure $x_{s,a}$ for π is the expected number of times action a is performed in state s before the goal is reached, when executing π from the initial state s_0.

This dual formulation can be interpreted as a probabilistic flow problem where one unit of flow is injected at the initial state (the source), transits via transient states, and reaches goal states (the sink). Constraint (C4) defines the flow exiting state s by applying actions prescribed by the policy, and constraint (C5) defines the flow entering state s by transiting from states s' via actions prescribed by the policy. Constraint (C6) captures the conservation of flow at transient states: the flow exiting each of these states must equal the flow entering it, with the exception of the initial state for which the exiting flow exceeds the entering flow by the one unit initially injected. Constraint (C7) enforces that all the flow reaches goal states, i.e., that the policy is safe. Finally, the objective function ensures that the total expected cost of the policy is minimized.

Let x^* be an optimal solution of the Dual LP, then the corresponding optimal stochastic policy is:

$$\pi^*(s,a) = \frac{x^*_{s,a}}{out(s)}$$

For SSPs, there always exist an optimal policy that is deterministic. Deterministic policies lie at the extreme points of the feasible region of the Dual LP, and any LP solver based on the simplex method will therefore return a deterministic optimal policy.[4] However, the ability of the Dual LP formulation to deal with stochastic policies makes it useful for handling more complex problems than SSPs, including problems with constraints, discussed in the next subsection, for which all optimal policies may be stochastic.

The Dual LP formulation can be easily adapted to infinite horizon and discounted reward maximization problems, as well as problems that go beyond SSPs such as that of finding a policy maximizing the probability of reaching the goal. For the latter, it suffices to remove constraint (C7) and replace the objective with that of maximizing the probability $\sum_{s \in S_g} in(s)$. The resulting LP is called the MAX-PROB LP.

Moreover, there are a number of options to handle unavoidable dead-ends. The first is the Finite Penalty method, where a dummy action that directly reaches the goal with probability 1 is applicable from every non-goal state and incurs a very high fixed cost. A more elegant and principled approach afforded by linear programming is to consider goal reachability probability and policy expected cost as two different objectives. For instance, one can use the Min-Cost given Max-Prob criterion which computes minimal cost policies amongst those with maximal goal reachability. This requires solving two LPs: the MAX-PROB LP to obtain the maximal goal reachability probability $pmax$, and then a slight variant of the Dual LP where the right hand-side of (C7) is replaced by $pmax$.

Despite being solvable in polynomial time, linear programs are however not competitive with other approaches for SSPs. The Primal LP has the same number of

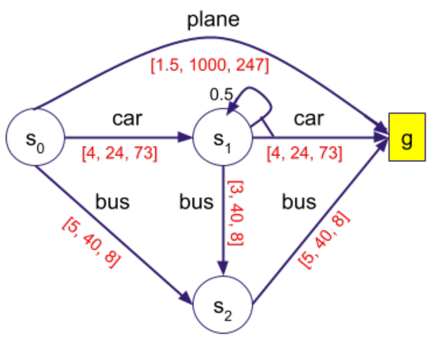
[4]More generally, a deterministic optimal policy for an SSP can be obtained from an stochastic optimal policy π^* for that SSP by deterministically selecting an action a at each state s such that $\pi^*(s,a) > 0$.

variables as Value Iteration, but in practice, its exact resolution is slower than the former. As for the Dual LP, its large number of variables $|S| \times |A|$ is a serious drawback. Therefore, it is only used to solve problems beyond SSPs which are not adequately covered by other approaches, such as constrained SSPs as we explain next.

9.4.2 Linear Programs for Constrained SSPs

Constrained SSPs, introduced in Section 8.3.3, are SSPs with multiple cost functions and constraints bounding their expected value. They are formally defined as follows.

Definition 9.21. A *constrained stochastic shortest path problem* (C-SSP) is a tuple $(\Sigma_{constr}, s_0, S_g)$, where $\Sigma_{constr} = (S, A, \gamma, \Pr, \vec{cost}, \vec{u})$, $S, A, \gamma, \Pr, s_0$ and S_g are defined as in an SSP, $\vec{\text{cost}} = [\text{cost}_0, \ldots, \text{cost}_k]$ is a vector of $k+1$ cost functions such that $\text{cost}_0 : S \times A \mapsto \mathbb{R}^+$ is the primary cost function and $\text{cost}_i : S \times A \mapsto \mathbb{R}$ for $i \in \{1, \ldots, k\}$ are the secondary cost functions, and $\vec{u} = [u_1, \ldots, u_k]$ is a vector of k upper-bounds on the secondary costs. □



min $1.5x_{0,p} + 4x_{0,c} + 4x_{1,c} + 5x_{0,b} + 3x_{1,b} + 5x_{2,b}$
subject to:

$$\begin{array}{lr}
x_{0,p} + x_{0,c} + x_{0,b} = 1 & (s_0\text{: C4, C5, C6}) \\
x_{0,p} + 0.5x_{1,c} + x_{2,b} = 1 & (g\text{: C5, C7}) \\
x_{1,c} + x_{1,b} - x_{0,c} - 0.5x_{1,c} = 0 & (s_1\text{: C4, C5, C6}) \\
x_{2,b} - x_{0,b} - x_{1,b} = 0 & (s_2\text{: C4, C5, C6}) \\
247x_{0,p} + 73(x_{0,c} + x_{1,c}) + & \\
\quad 8(x_{0,b} + x_{1,b} + x_{2,b}) \le 184 & (\text{CO}_2\text{: C8}) \\
1000x_{0,p} + 24(x_{0,c} + x_{1,c}) + & \\
\quad 40(x_{0,b} + x_{1,b} + x_{2,b}) \le 260 & (\text{money: C8})
\end{array}$$

Figure 9.7. A simple travel C-SSP (left) and its Dual LP (right).

Example 9.22. Consider the simple C-SSP on the left-hand side of Figure 9.7. Alex needs to frequently travel from home (at s_0) to visit his family (represented by the goal g). For each trip, he can take the plane, which takes just 1.5h but costs 1000 euros and consumes 247 kg of CO_2. He can alternatively take the bus, changing roughly mid way (at s_2); this only costs 40 euros per leg and consumes 8kg of CO_2, but each leg takes 5 hours. Finally, he can take a combination of dirt and mountain roads with his car, which is in principle faster and cheaper than the bus, and consumes much less CO_2 than the plane. However, the road is frequently closed (50% of the time) due to fires, flood, excessive snow, or accidents, and he is often forced to go back to the only town along the way at s_1 and stay there pending the issue being resolved before resuming his trip. Alternatively, from s_1, he can catch a bus to g via s_2. The first four constraints in the right-hand side of the figure (those labelled s_0 to s_2) are the Dual LP constraints capturing the underlying SSP. The occupation measures are $x_{i,j}$ where $i \in \{0, 1, 2\}$ represents the index of the state, and $j \in \{b, c, p\}$ represents the action of taking the bus, car, and plane, respectively.

The vectors in red in the figure represent the cost vectors $\vec{cost}(s,a)$ associated with taking action a in state s. Alex would like to minimize the time spent traveling, and this is therefore the primary cost. This is reflected in the objective of the dual LP in the figure. Money and CO_2 consumption are the secondary costs. Alex has calculated that he can afford an average of 260 euros (each way) per visit. Moreover, he is not prepared to increase his carbon footprint by more than 184 kg of CO_2 per visit on average. Hence the bounds vector is $\vec{u} = [260, 184]$. □

A solution to a C-SSP is a stochastic policy minimizing the expected primary cost, subject to the expected secondary costs being below their respective upper bounds. Not all C-SSPs have solutions, as the constraints may be individually or mutually unsatisfiable. In the following we write $V_c^\pi(s)$ for the value function of policy π at state s for the cost function c.

Definition 9.23. A *solution to a C-SSP* $(\Sigma_{constr}, s_0, S_g)$ is a safe stochastic policy minimizing $V^\pi_{\text{cost}_0}(s_0)$ under the constraints that $V^\pi_{\text{cost}_i}(s_0) \le u_i \; \forall i \in \{1,\ldots,k\}$. □

It is important to notice that the constraints apply to the *expected* values $V^\pi_{\text{cost}_i}(s_0)$ at the *initial state*. These constraints do not apply to states other than the initial one even in expectation, nor to individual executions of the policy from the initial state. For instance, if $cost_i$ measures travel time and u_i represents a deadline, there is no guarantee that all possible executions of the policy meet the deadline. Providing stronger guarantees would require augmenting the state space with state variables representing the accumulated cost for the various cost functions, leading to an increase of the size of the state space exponential in k. Irrespective of the high complexity, the benefits of augmenting the state space in this way are questionable, as there will always be extremely unlikely executions of the policies that exceed any reasonable fixed cost bound. The advantages of expected cost constraints are that they do not increase the theoretical complexity of the problem and can be handled with minimal changes to the Dual LP.

Indeed, C-SSPs have the same worst-case time complexity as SSPs, i.e., polynomial in $|S|\times|A|$ (and hence exponential time in the size of the factored MDP representation). The Dual LP for C-SSPs (Algorithm 9.16) only requires one additional constraint, namely (C8), which bounds the expected secondary costs $V^\pi_{\text{cost}_i}(s_0)$. If the C-SSP has a solution, this LP's optimal solution is an optimal set of occupation measures x^* from which an optimal stochastic policy π^* can be retrieved. Otherwise the LP solver returns that the problem has no solution.

Paradoxically, optimal deterministic policies, in addition to not being as good as stochastic ones, are more expensive to compute, making the problem NP-hard. In practice, they require additional constraints that involve new binary variables, turning the LP into a Mixed Integer Program (MIP).

Example 9.24. We continue the example depicted in Figure 9.7. The last two constraints on the right-hand side enforce the bounds (C8) on the secondary money and CO_2 costs of the Dual LP for C-SSPs. The optimal stochastic policy obtained by solving the Dual LP for C-SSPs uses the plane and the bus roughly 20% of the time each, and the car the remaining 60% of the time. It never takes the bus from s_1 to

```
Dual LP for C-SSPs($\Sigma_{const}, s_0, S_g$)
    min $\sum_{s \in S \setminus S_g, a \in A(s)} \text{cost}_0(s,a) x_{s,a}$

        s.t. (C3) - (C7)
             $\sum_{s \in S \setminus S_g, a \in A(s)} \text{cost}_i(s,a) x_{s,a} \leq u_i \qquad \forall i \in \{1, \dots, k\}$     (C8)
```

Algorithm 9.16. Dual Linear Program for Constrained SSPs

s_2. On average, it takes 8.3h to reach the goal, and it reaches the bounds of 260 euros and 184 kg of CO_2 exactly. In contrast, the optimal deterministic policy takes the bus all the way from s_0 to g, leading to a journey of 10h, a cost of 80 euros, and a CO_2 consumption of 16 kg. □

9.4.3 Hybrid LP and Heuristic Search for Constrained SSPs

The size of the occupation measure space is the product $|S| \times |A|$. It is too large for the Dual LP for C-SSPs to be practical. In realistic cases, it would require solving linear programs with millions or even billions of variables. However, by hybridizing linear programming and heuristic search, it is possible to guide the search for a solution in such a way as to explore only a small fraction of the occupation measure space.

i-dual is such a hybrid algorithm for C-SSPs. Recall that heuristic search algorithms of Section 9.2 explore progressively larger envelopes rooted at the initial state, evaluate fringe states using a heuristic function, and stop when the initial state is "solved". At each iteration, they may expand fringe states reachable under the current policy, or perform Bellman updates on reachable interior states. i-dual is similar except that it optimally solves the Dual LP for C-SSPs on the current envelope at each iteration instead of performing Bellman backups. It expands all fringe states reachable under the optimal policy found, and stops iterating whenever all non-interior states of the policy are actual goal states. Using linear programming as a subroutine allows i-dual to handle constraints and produce stochastic policies whereas previous heuristic search algorithms could not.

i-dual uses admissible heuristics (lower bounds on V_c^*) for each cost function c, whether primary or secondary. The primary heuristic serves the usual purpose of guiding the search towards cheap safe policies, whereas the secondary heuristics help with early pruning of regions of the policy space that do not satisfy the constraints.

More formally, let $E = I \cup F \cup G$ be the current envelope explored by i-dual where I are the interior states, F the fringe states, and $G \subseteq S_g$ the goal states in the envelope, and let $\vec{h} = [h_0, \dots, h_k]$ be a vector of $k+1$ heuristic functions such that $h_i(s) \leq V^*_{\text{cost}_i}(s)$ for all $i \in \{0, \dots, k\}$ and state $s \in S$. Let the set of envelope actions be $A_E = \{a \in A(s) \mid s \in I\}$. At each iteration, i-dual solves the partial C-SSP $((E, A_E, \gamma, \Pr, \vec{cost}, \vec{u}), s_0, F \cup G)$, using the heuristics given by $\vec{h}$ at fringe states.

The LP solved by i-dual at each iteration is shown in Algorithm 9.17.

```
i-dual LP$(((E, A_E, \gamma, \Pr, \vec{cost}, \vec{u}), s_0, F \cup G), \vec{h})$
    $I \leftarrow E \setminus (F \cup G)$
    min $\sum_{s \in I, a \in A_E(s)} \text{cost}_0(s,a) x_{s,a} + \sum_{s \in F} h_0(s) in(s)$

      s.t. (C3) - (C7) [replacing $S \setminus S_g$ with $I$, $S$ with $E$, $S_g$ with $F \cup G$, $A$ with $A_E$]
        $\sum_{s \in I, a \in A_E(s)} \text{cost}_i(s,a) x_{s,a} + \sum_{s \in F} h_i(s) in(s) \leq u_i \quad \forall i \in \{1, \ldots, k\}$
                                                                                (C9)
```

Algorithm 9.17. Linear Program Solved by i-dual at Each Iteration

```
i-dual$((\Sigma_{constr}, s_0, S_g), \vec{h})$
    $E \leftarrow \{s_0\}$; $F \leftarrow \{s_0\}$; $G \leftarrow \varnothing$; $A_E \leftarrow \varnothing$; $F_\pi \leftarrow \{s_0\}$
    while $F_\pi \neq \varnothing$ do                  // As long as E has reachable fringe states
        foreach $s \in F_\pi$ do                        // expand reachable fringe states
            remove $s$ from $F$                 // and build new partial problem
            $A_E \leftarrow A_E \cup Applicable(s)$          // by updating E, $A_E$, G, and F
            foreach $a \in Applicable(s)$ and $s' \in \gamma(s,a)$ do
                if $s' \notin E$ then
                    add $s'$ to E
                    if $s' \in S_g$ then
                        add $s'$ to $G$ else
                            add $s'$ to $F$

        $x \leftarrow$ Solve i-dual LP$(((E, A_E, \gamma, \Pr, \vec{C}, \vec{u}), s_0, F \cup G), \vec{h})$    // Solve LP
        if the LP was not solvable then
            return Unsolvable
        $F_\pi \leftarrow \{s \in F | in(s) > 0\}$             // and update reachable fringe states
    foreach $(s,a) \in E \times A_E$ such that $x_{s,a} > 0$ do
        $\pi(s,a) \leftarrow x_{s,a}/out(s)$
    return $\pi$
```

Algorithm 9.18. i-dual, a hybrid algorithm for C-SSPs.

i-dual (Algorithm 9.18) uses this LP as a subroutine. It starts from the envelope E containing only the initial state s_0 and from the empty policy. At each iteration, it updates the envelope by expanding all fringe states F_π reachable under the current policy, and builds a new partial problem over the updated envelope, which is then solved with i-dual LP. When the current policy has no fringe states (i.e. all terminal states of the policy are goal states), then i-dual has found an optimal stochastic policy for the original C-SSP. If any of the calls to i-dual LP fails, then the original C-SSP is unsolvable. Observe that i-dual runs in polynomial time: each iteration (including i-dual LP) runs in polynomial time, and there are at most linearly many iterations in the number of states of the C-SSP.

Theorem 9.25. *Given a C-SSP $C = (\Sigma_{constr}, s_0, S_g)$ and a vector of admissible heuristics $\vec{h}$,* i-dual *returns an optimal stochastic policy π for C if C is solvable, and returns Unsolvable otherwise. It does so in polynomial time in $|S| \times |A|$.*

It also possible to use i-dual with inadmissible heuristics to speed up the resolution of the LP. If the primary cost heuristic h_0 is not admissible, then i-dual remains correct and complete, i.e. it will return a safe stochastic policy that satisfies the constraints if one exists, but the policy may be suboptimal. If a secondary heuristic is not admissible, then i-dual will be correct but possibly incomplete, i.e, it may deem the problem unsolvable even if a safe policy satisfying the constraints exists. This is because constraint (C9) may unduly prune solutions.

A vector of admissible heuristics $\vec{h}$ can be obtained by constructing an individual heuristic h_i for each cost function $cost_i$, e.g., via one of the approaches of Section 9.3. This however has the downside of not taking into account possible dependencies between the cost functions.

Let us introduce a heuristic that allows us to simultaneously reason over all the cost functions. Consider first a single cost function $cost$. The *projection occupation-measure heuristic* h^{pom} combines into a single LP the Dual LP for the syntactic projections $\Sigma_{\{y\}}$ over all state variables y (see Section 9.3.4). Let $x^y_{v,a}$ be the occupation-measure variable belonging to the state $\{y = v\}$ in the syntactic projection onto $\{y\}$ and action a. Heuristic h^{pom} connects the different projections, forcing the LP solution to execute in all projections all actions exactly the same expected number of times, by including the following *tying constraints*:

$$\sum_{\text{value } v \text{ of } y} x^y_{v,a} = \sum_{\text{value } v' \text{ of } y'} x^{y'}_{v',a}, \tag{9.12}$$

for all actions a and pairs of distinct state variables y, y'.

The LP's objective function is the objective function of the Dual LP for the syntactic projection onto some $\hat{y}$. The precise choice of $\hat{y}$ does not matter given (9.12). $h^{\text{pom}}(s)$ gives the optimal objective value. To see that this is an admissible estimate of $V^*_{cost}(s)$, let $x^*_{s,a}$ be an optimal solution of the Dual LP for the problem (Σ, s, S_g) and cost function $cost$. Recall that the objective value corresponding to x^* is equal to $V^*_{cost}(s)$, and consider the assignment:

$$\hat{x}^y_{v,a} = \sum_{s' \in S:\, s' \text{ contains } y=v} x^*_{s',a}$$

for all variables y, values v of y, and actions a. As x^* satisfies the constraints of the Dual LP for the concrete problem, $\hat{x}$ necessarily satisfies the Dual LP for all the projections. Moreover, $\hat{x}$ satisfies the tying constraints by construction. Since $\hat{x}$ induces an objective value equal to that of x^*, it follows that $h^{\text{pom}}(s) \leq \textit{objective value of } \hat{x} = V^*_{cost}(s)$.

i2-dual is an enhanced variant of i-dual, which embeds the projection occupation-measure LP directly into the i-dual LP in place of the external heuristics $\vec{h}$. This leads policy update and heuristic computation to work in unison. To obtain an admissible estimate of the expected cost under $cost_i$ starting from the fringe states, the only change required is synchronizing the flow-surplus in the flow-conservation constraints (C6) of the projections with the in-flow of the fringe states. Using the same projection occupation-measure variables for all cost functions creates a tight link between all the heuristic estimates.

9.5 Online Probabilistic Approaches

Often, finding a complete plan then acting according to that plan is often not a feasible nor a desirable approach. It is not feasible for complexity reasons in large domains, that is, a few dozens ground state variables. Even with good heuristics, algorithms seen in Section 9.2 cannot always address large domains, unless the designer is able to carefully engineer and decompose the domain. Even memorizing a safe policy as a table lookup in a large domain is by itself challenging to current techniques (that is, decision diagrams and symbolic representations). However, a large policy contains necessarily many states that have a very low probability of being reached, e.g., lower than the probability of unexpected events not modeled in Σ. These improbable states may not justify being searched, unless they are critical. They can be further explored if they are reached or become likely to be reached while acting.

Furthermore, even when heuristic planning techniques do scale up, acting is usually time-constrained. A trade-off between the quality of a solution and its computing time if often desirable, for example, there is no need to improve the quality of an approximate solution if the cost of finding this improvement exceeds its benefits. Such a trade-off can be achieved with an online anytime algorithm that computes a rough solution quickly and improves it when given more time.

Finally, the domain model is seldom precise and complete enough to allow for reliable long-term plans. Shorter lookaheads with progressive reassessments of the context are often more robust. This is often implemented in a receding horizon scheme, which consists in planning for d steps towards the goal, performing one or a few actions according to the found plan, then replanning further.

This section presents a few techniques that perform online lookaheads and permit to interleave planning and acting in probabilistic domains. These techniques are based on a general schema, discussed next.

9.5.1 Lookahead Methods

Lookahead methods allow an actor to progressively elaborate its deliberation while acting. They rely on a procedure such as MDP-Lookahead presented earlier (Algorithm 8.2), and a generative sampling function. A full definition of $\gamma(s,a)$ for all $a \in Applicable(s)$ is not necessary to a partial exploration. Most partial exploration techniques rely on sampling methods. They search only one or a few random outcomes in $\gamma(s,a)$ over a few actions in *Applicable*(s).

Definition 9.26. A *generative sampling model* of a domain $\Sigma = (S, A, \gamma, \Pr, \mathrm{cost})$ is a stochastic function

$$\mathsf{Sample} : S \times A \to S \times \mathbb{R}, \text{ such that: } \mathsf{Sample}(s,a) = (s', \mathrm{cost}(s,a,s')),$$

where $s' \in \gamma(s,a)$ is randomly distributed according to $\Pr(s'|s,a)$. □

We assume that several calls to Sample returns a set of states s' that independently and identically distributed (the classical *i.i.d* assumption). Note a domain can be defined by specifying S, A and a generative Sample function. One does not need γ and *a priori* estimates of the probability and cost distributions of Σ. A domain simulator is generally the way to implement the function Sample, which provides an implicit (also referred to as a model-free) specification of an MDP.

Approaches and properties of *Lookahead*. One possible option is to memorize the search space explored progressively: each call to *Lookahead* relies on knowledge acquired from previous calls; its outcome augments this knowledge. As an alternative to this *memory-based* approach, a *memoryless* strategy would start with a fresh look at the domain in each call to *Lookahead*. The choice between the two options depends on how stationary the domain is, how often an actor may reuse its past knowledge, how easy it is to maintain this knowledge, and how this can help improve the behavior.

The advantages of partial lookahead come naturally with a drawback, which is the lack of a guarantee on the optimality and safety of the solution. Indeed, it is not possible in general to choose $\pi(s)$ with a bounded lookahead while being sure that it is optimal, and, if the domain has dead ends, that there is no dead end descendant in $\widehat{\gamma}(s,\pi)$. Finding whether a state s is unsafe may require in the worst case a full exploration of the search space starting at s. In the bounded lookahead approach, optimality and safety are replaced by a requirement of bounds on the distance to the optimum and on the probability of reaching the goal. In the memory-based approaches, one may also seek asymptotic convergence to safe and/or optimal solutions.

Three approaches to the design of a *Lookahead* procedure are presented next:

- domain determinization and replanning with deterministic search,
- stochastic simulation, and
- sparse sampling and Monte Carlo planning techniques.

The last two approaches are interfaced with a generative sampling model of Σ using a Sample function: they do not need *a priori* specification of probability and cost distributions. The third one is also memoryless; it is typically used in a receding horizon scheme. However, many algorithms implementing these approaches can be used for online interleaved planning and acting framework, as well as for offline planning. Their control parameters allow for a continuum from the computation of a greedy policy computed at each state to a full exploration and definition of $\pi(s_0)$.

9.5.2 Planning with Deterministic Search

Determinization techniques. The idea here is to use any deterministic planner to generates a path π_d from the current state to a goal for the most probable outcomes determinized domain, then to act using π_d until reaching a state s that is not in the domain of π_d. At that point one generates a new deterministic plan starting at s.

Note, however, that this approach does not cope adequately with dead ends. Even if the deterministic planner is complete and finds a path to the goal when there is one, executing that path may lead along a nondeterministic branch to an unsafe state.

RFF (Algorithm 9.19) relies on a deterministic planner, called Det-Plan, to find in Σ_d an acyclic path from a state to a goal. Σ_d takes in Σ the most probable $s' \in \gamma(s,a)$. Procedure Det-Plan returns a path Σ_d taken as a partial policy. RFF memorizes previously generated deterministic paths and extends them for states that have a high reachability probability. RFF can be used as an online *Lookahead* procedure, possibly with additional control parameters, or as an an offline planner. In the latter case, RFF repeatedly extends undefined branches in $\widehat{\gamma}(s_0,\pi)$.

```
RFF($\Sigma, s_0, S_g, \theta$)
    $\pi \leftarrow$ Det-Plan($\Sigma_d, s_0, S_g$)
    while $\exists s \in \widehat{\gamma}(s_0,\pi)$ such that
            $[(\pi(s)$ is undefined$) \wedge (s \notin S_g) \wedge (\Pr(s|s_0,\pi) \geq \theta)]$ do
        $\pi \leftarrow \pi \cup$ Det-Plan($\Sigma_d, s, S_g \cup$ Targets($\pi, s$))
```

Algorithm 9.19. RFF, a determinization planning algorithm.

RFF initializes the policy π with the pairs (state, action) corresponding to a deterministic plan from s_0 to a goal, then it extends π. It looks for a fringe state $s' \in \widehat{\gamma}(s_0,\pi)$ that has a successor s not in S_g and for which π is undefined. If the probability of reaching s is above some threshold θ, RFF extends π with another deterministic path from s to a goal or to another state already in the domain of π. The set of additional goals given to Det-Plan, denoted Targets(π, s), can be the already computed *Domain*(π) or any subset of it. If the entire *Domain*(π) is too large, the overhead of using it in Det-Plan can be larger than the benefit of reusing paths already planned in π. A trade-off reduces Targets(π, s) to k states already in the domain of π. These can be taken randomly in *Domain*(π) or chosen according to some easily computed criterion.

Computing $\Pr(s|s_0,\pi)$ can be time-consuming (a search and a sum over all paths from s_0 to s with π). This probability can be estimated by sampling. A number of paths starting at s_0 following π are sampled; this allows the estimation of the total probability of reaching non-goal states that are not in the domain of π. RFF terminates when this frequency is lower than θ.

Algorithm 9.19 requires a domain without reachable dead ends. However, RFF can be extended to domains with avoidable dead ends, that is, where s_0 is safe. This is achieved by introducing a backtrack point in a state s which is either an immediate dead end or for which Det-Plan fails. That state is marked as unsafe; a new search starting from its predecessor s' is attempted to change $\pi(s')$ and avoid the previously failed action.

RFF algorithm does not attempt to find an optimal or near optimal solution. However, the offline version of RFF finds a *probabilistically safe* solution, in the sense that the probability of reaching a state not in the domain of π, either safe or unsafe, is upper bounded by θ.

Mixed Deterministic-Probabilistic Approaches. In Section 8.3.4 we discussed modeling domains where all but a few of the actions are deterministic, by mixing deterministic and nondeterministic approaches.

Here is a possible approach for planning in a domain that has both deterministic and nondeterministic actions. Assume that while planning from a current state s to a goal, the algorithm finds at some point a sequence $\langle(s, a_1), (s_2, a_2), \ldots, (s_{k-1}, a_{k-1}), (s_k, a)\rangle$ such that actions a_1 through a_{k-1} are deterministic, but a is nondeterministic. It is possible to compress this sequence to a single nondeterministic step (s, a), the cost of which is the sum of cost of the k steps and the outcome $\gamma(s, a)$ of which is the outcome of the last step. This idea can be implemented as sketched in Algorithm 9.20. Its advantage is to focus the costly processing on a small part of the search space.

```
Incremental-compression-and-search($\Sigma, s_0, S_g$)
    while there is an unsolved state $s$ in current $\widehat{\gamma}(s_0, \pi)$ do
        search for an optimal path from $s$ to a goal
                until a nondeterministic action $a$
        compress this path to a single nondeterministic step
        $\pi(s) \leftarrow a$
        revise with Bellman-Update
```

Algorithm 9.20. Incremental-compression-and-search for sparse probabilistic domains.

The notion of mixed deterministic-probabilistic domains can be extended further to cases in which $|\gamma(s, a)| < k$ and $Applicable(s) < m$ for some small constants k and m. Sampling techniques, discussed next, are particularly useful in these cases.

9.5.3 Stochastic Simulation Techniques

Stochastic simulation techniques rely on a generative Sample function. They run simulated walks from s_0 to a goal along best current actions by sampling one outcome for each action. Algorithms implementing this idea are inspired from LRTA* [639]. They can be implemented as offline planners or online *Lookahead* procedures.

One such algorithm, called RTDP, runs a series of simulated trials starting at s_0. A trial performs a Bellman update on the current state, then it proceeds to a randomly selected successor state along the current action $\pi(s)$, that is, from s to some random $s' \in \gamma(s, \pi(s))$. A trial finishes when reaching a goal. The series of trials is pursued until either the residual condition is met, which reveals near convergence, as in Find&Revise, or the amount of time for planning is over. At that point, the best action in s_0 is returned. With these assumptions RTDP is an anytime algorithm.

If a goal is reachable from *every* state in the search space and if the heuristic V_0 is admissible then every trial reaches a goal in a finite number of steps and improves the values of the visited states over the previous values. Hence, RTDP converges asymptotically to V^*, but not in a bounded number of trials. Note that these assumptions are stronger than the existence of a safe policy.

```
LRTDP($\Sigma, s_0, g, V_0$)
    until $s_0$ is solved or planning time is over do
        LRTDP-Trial($s_0$)

LRTDP-Trial($s$)
    $visited \leftarrow$ empty stack
    while $s$ is unsolved do
        push($s, visited$)
        Bellman-Update($s$)
        $(s, c) \leftarrow$ Sample($s, \pi(s)$)
    $s \leftarrow$pop($visited$)
    while Check-Solved($s$) is true and $visited$ is not empty do
        $s \leftarrow$pop($visited$)
```

Algorithm 9.21. LRTDP algorithm.

Algorithme LRTDP (for *Labelled RTDP*), improves over RTDP by explicitly checking and labeling solved states. LRTDP avoids visiting solved states twice. It calls LTRDP-Trial(s_0) repeatedly until planning time is over or s_0 is solved. A trial is a simulated walk along current best actions. It stops when reaching a solved state. A state s visited along a trial is pushed in a stack $visited$. When needed, it is expanded and Bellman updated. The trial is pursued on a randomly generated successor of s. The procedure Sample (s, a) returns a pair $(s', cost(s, \pi(s), s')$, with $s' \in \gamma(s, \pi(s))$ randomly drawn according to the distribution $\Pr(s'|s, \pi(s))$.

The states visited along a trial are checked in LIFO order using the procedure Check-Solved to label them as solved or to update them. Check-Solved(s) searches through $\widehat{\gamma}(s, \pi)$ looking for a state whose residual is greater than the margin η. If it does not find such a state ($flag$ = true), then there is no open state in $\widehat{\gamma}(s, \pi)$: s and its descendants in $\widehat{\gamma}(s, \pi)$ (kept in the $closed$ list) are labeled as solved. Otherwise, there are open states in $\widehat{\gamma}(s, \pi)$. The procedure does not explore further down the successors of an open state (its residual is larger than η); it continues on its siblings.

When all the descendants of s whose residual is less or equal to η have been examined (in that case $open = \varnothing$), the procedure tests the resulting *flag*. If s is not yet solved (that is, *flag*= false), a Bellman update is performed on all states collected in *closed*. Cycles in the *Envelope* are taken care of (with the test $s' \notin open \cup closed$): the search is not pursued on successors that have already been met. The complexity of Check-Solved(s) is linear in the size of the *Envelope*, which may be exponential in the size of the problem description.

Note that by definition, goal states are solved; hence the test "s is unsolved" in the

```
Check-Solved($s$)
    $flag \leftarrow$ true
    $open \leftarrow closed \leftarrow$ empty stack
    if $s$ is unsolved then push($s, open$)
    while $open$ is not empty do
        $s \leftarrow$ pop($open$)
        push($s, closed$)
        if $|V(s) - Q(s, \pi(s))| > \eta$ then $flag \leftarrow$ false
        else
            foreach $s' \in \gamma(s, \pi(s))$ do
                if $s'$ is unsolved and $s' \notin open \cup closed$ then push($s', open$)
    if $flag$= true then
1       foreach $s' \in closed$ do label $s'$ as solved              // labeling step
    else
        while $closed$ is not empty do
            $s \leftarrow$ pop($closed$)
            Bellman-Update($s$)
    return $flag$
```

Algorithm 9.22. Check-Solved, procedure to check and label solve states for LRTDP.

two preceding procedures checks the explicit labeling performed by Check-Solved (*labeling step*) as well as the goal condition.

If a goal is reachable from every state and V_0 is admissible, then LRTDP-Trial always terminates in a finite number of steps. Furthermore, if the heuristic V_0 is admissible and monotone, then the successive values of V with Bellmann updates are nondecreasing. Under these assumptions, each call to Check-Solved(s) either labels s as solved or increases the value of some of its successors by at least η while decreasing the value of none. This leads to the same complexity bound as Value Iteration:

Proposition 9.27. LRTDP *with an admissible and monotone heuristic on a problem where a goal is reachable from every state converges in a number of trials bounded by* $1/\eta \sum_S [V^*(s) - V_0(s)]$.

This bound is mainly of theoretical interest. Of more practical value is the anytime property of LRTDP: the algorithm produces a good solution that it can improve if given more time or in successive calls in MDP-Lookahead. Because Sample returns states according to their probability distribution, the algorithm solves frequent states faster than on less probable ones. As an offline planner (that is, repeated trials until s_0 is solved), its practical performances are comparable to those of the other heuristic algorithms presented earlier.

9.5.4 Sampling and Monte Carlo Approaches

The stochastic simulation approach of the previous section with a generative Sample function can be extended and used in many ways, in particular with the bounded walks and sampling strategies discussed here.

Let π_0 be an arbitrary policy, used at initialization. For example, $\pi_0(s)$ is the greedy policy, locally computed only when needed as $\pi_0(s) = \text{argmin}_a Q^{V_0}(s, a)$ for some heuristic V_0. If the actor has no time for planning, then $\pi_0(s)$ is the default action. If it can afford a lookahead, then it can improve π_0 with Monte Carlo rollouts.

Monte Carlo Rollout. Let us use the Sample procedure to simulate a random bounded walk in the search tree down to a depth d. The first step is a chosen action a applicable in s; the remaining $d - 1$ steps follow a initial policy π_0. Let $\sigma^d_{\pi_0}(s, a) = \langle s, s_1, s_2, \dots, s_d\rangle$ be the sequence of states visited during this walk, with $(s_1, c_1) \leftarrow \text{Sample}(s, a)$ and $(s_{i+1}, c_{i+1}) \leftarrow \text{Sample}(s_i, \pi_0(s_i))$ for $1 \leq i < d$. This history $\sigma^d_{\pi_0}(s, a)$ is called a *rollout* for a in s with π_0. Let $Q^d_{\pi_0}(s, a)$ be the cost-to-go of a in s as estimated by this rollout. It can computed by a call to the procedure Rollout (s, a, π_0, d), which stops after d steps, estimating the remaining cost with V_0, or when reaching a goal (Algorithm 9.23).

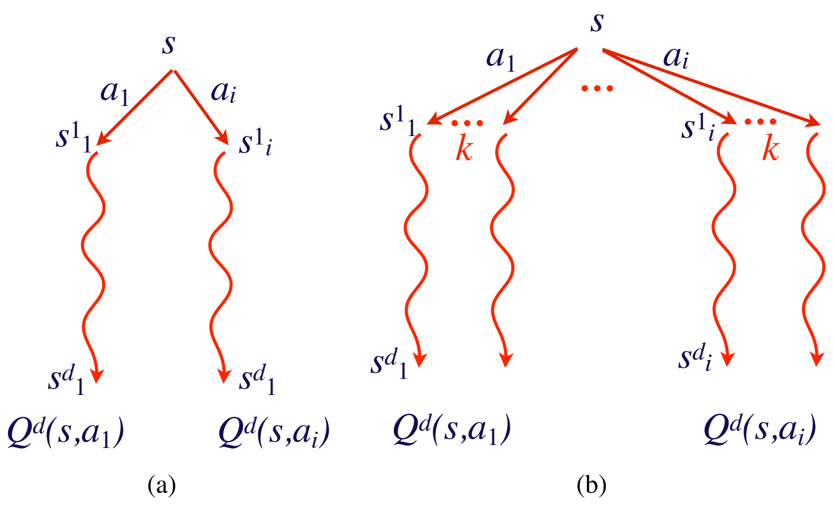


Figure 9.8. (a) Single Monte Carlo rollouts for actions in s; (b) Multiple rollouts for actions in s.

Let us perform a rollout for every action applicable in s following π_0, as depicted in Figure 9.8(a), and let us define a new policy:

$$\pi(s) = \underset{a}{\text{argmin}}\, Q^d_{\pi_0}(s, a).$$

The argument of Proposition 9.1 applies here: policy π dominates the base policy π_0.

However, a single rollout of a in s will not give a good estimate of the cost-to-go. A MultipleRollout procedure performs k similar simulated walks of d steps for each

```
Rollout($s, a, \pi, d$)
    $(s', c) \leftarrow$ Sample($s, a$)                    // where $c = \mathrm{cost}(s, a, s')$
    return $[c +$ Rest-Rollout$(s', \pi, d - 1)]$

Rest-Rollout($s, \pi, d$)
    if $s \in S_g$ then return 0
    else if $d = 0$ then return $V_0(s)$
    else
        $(s', c) \leftarrow$ Sample($s, \pi(s)$)              // $c = \mathrm{cost}(s, \pi(s), s')$
        return $[c +$ Rest-Rollout$(s', \pi, d - 1)]$
```

Algorithm 9.23. Performing a rollout for a in s with π and computing its total cost

applicable a in s (Algorithm 9.24 and Figure 9.8(b)). The average of their costs assesses $Q(s, a)$. This is the case since Sample returns random states according to the probability distributions. This procedure is *probabilistically approximately correct*, that is, it provides a probabilistically safe solution (not guaranteed to be safe) whose distance to the optimum is bounded. In each s, it performs $|Applicable(s)| \times k \times d$ calls to Sample.

```
MultipleRollout($s, \pi, k, d$)
    foreach $a \in Applicable(s)$ do
        $Q(s, a) \leftarrow 0$
        for $k$ times do
            $Q(s, a) \leftarrow Q(s, a) + \frac{1}{k}$Rollout$(s, a, \pi, d)$
    $\pi(s) \leftarrow \mathrm{argmin}_a\, Q(s, a)$
```

Algorithm 9.24. MultipleRollout, a multiple rollout procedure

Note the similarity of MultipleRollout to the Policy Iteration procedure: in both cases we first compute the V^π, or equivalently the Q^π functions for a given π, then we improve π with the newly computed value or cost functions. There two main differences: *(i)* Policy Iteration computes V^π systematically while MultipleRollout estimates with sampling an approximation of Q^π, and *(ii)* Policy Iteration defines π over all S, whereas here we do it incrementally for current s. The already mentioned *approximate policy iteration* techniques rely on MultipleRollout performed over a number of representative states, and a generalization of the resulting policy to S with a learning procedure (see Chapter 10).

Sparse Sampling. We can extend the previous approach with bounded multiple rollouts in s and recursively in each of its descendants reached by these rollouts.

Procedure SLATE builds a tree in which nodes are states; arcs correspond to transitions to successor states, which are randomly sampled. Two parameters d and k

bound the tree, respectively in depth and sampling width (see Figure 9.9). At depth d, a leaf of the tree gets a heuristic value estimated by V_0. In an interior state s and for each action a applicable in s, k successors are randomly sampled. The average of their estimated values over the set of pairs $(s', \text{cost}(s, a, s') \in samples$ is used to compute recursively the cost-to-go $Q(a, s)$. The minimum over all actions in $Applicable(s)$ gives $\pi(s)$ and $V(s)$, as in Bellman-Update.

```
SLATE($s, d, k$)
    if $d = 0$ then return $V_0(s)$
    if $s \in S_g$ then return 0
    foreach $a \in Applicable(s)$ do
        $samples \leftarrow \varnothing$
        for $k$ times do
            $samples \leftarrow samples \cup$ Sample$(s, a)$
        $Q(s, a) \leftarrow \frac{1}{k} \sum_{(s',c) \in samples} c +$ SLATE$(s', d-1, k)$
    $\pi(s) \leftarrow \text{argmin}_a \{Q(s, a)\}$
    return $Q(s, \pi(s))$
```

Algorithm 9.25. SLATE, sampling lookahead tree.

Assuming that a goal is reachable from every state, SLATE has the following properties:

- It does not require probability distributions: recall that successive calls to Sample (s, a) returns states in $\gamma(s, a)$ distributed according the $\Pr(s'|s, a)$, which allows estimating $Q(s, a)$.
- It defines a near-optimal policy: the difference $|V(s) - V^*(s)|$ can be bounded as a function of d and k.
- It runs in a worst-case complexity independent of $|S|$, but nonetheless exponential in $O((\alpha k)^d)$, where $\alpha = \max |Applicable(s)|$.

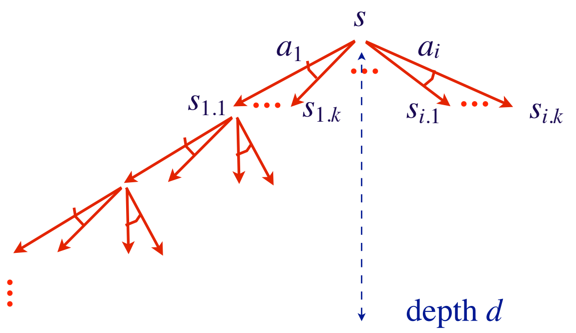


Figure 9.9. SLATE's sparse sampling tree.

Note the differences between SLATE and MultipleRollout: the latter is polynomial in d, but its approximation is probabilistic. SLATE provides a guaranteed approximation, but it is exponential in d. More precisely, SLATE returns a solution whose distance to the optimal policy is upper bounded $|V(s) - V^*(s)| < \epsilon$; it runs in $O(\epsilon^{log \epsilon})$.

A few improvements can be brought to SLATE. One may reduce the sampling with the depth of the state: the deeper is a state, the less influence it has on the cost-to-go of the root. Further, the data structure *samples* can implemented as a set with counters on its elements such as to perform a single recursive call on a successor s' of s that is sampled more than once. Note that the sampling width k can be chosen independently of $|\gamma(s,a)|$. However, when $k > |\gamma(s,a)|$, further simplifications can be introduced, in particular for deterministic actions. Finally, it is easy to refine SLATE into an anytime algorithm: an iterative deepening scheme with caching increases the horizon d until acting time (see Exercise 9.16).

9.5.5 MCTS and UCT

In SLATE, as well as in MultipleRollout, the sampling strategy is systematic. All actions in $Applicable(s)$ are explored in the same way. A sampling strategy would allow further exploring a promising action; it would prune out rapidly inferior options, but no action should be left untried. It would seek a trade-off between the number of times an action a has been sampled in s and the value $Q(s,a)$. This trade-off seeks to find a probably good solution while minimizing the search.

Monte Carlo Tree Search techniques. Monte Carlo Tree Search (MCTS) techniques, illustrated in Algorithm 9.26, rely on such a sampling strategy. MCTS is an online version of Find&Revise. It develops a focused part of the search space, starting from a current root state s_r, and performs Bellman updates within this focused part. It has a *find* step seeking an open state, and a *revise* step. However MCTS uses rollouts instead of heuristic estimates for the *revise* steps.

```
MCTS($s_r, d$)
   until termination condition do
1      select an open state $s \in \widehat{\gamma}(s_r, \pi)$
2      choose an action $\tilde{a} \in Applicable(s)$
3      $Q(s,\tilde{a}) \leftarrow [N(s,\tilde{a}) \times Q(s,\tilde{a}) + \text{Rollout}(s,\tilde{a},\pi_r,d)]/(1 + N(s,\tilde{a}))$
4      $N(s,\tilde{a}) \leftarrow N(s,\tilde{a}) + 1$
5      $\pi(s) \leftarrow \text{argmin}_a\{Q(s,a)\}$
6      update all ancestors of $s$ in $\widehat{\gamma}(s_r, \pi)$
   return $\pi(s_r)$ and $Q(s_r, \pi(s_r))$
```

Algorithm 9.26. MCTS, a procedure for MDP And/Or graphs

MCTS starts with some initial cost-to-go function Q_0. As in Find&Revise, a state is open when either it is a fringe or when it requires further updates. This latter condition is however assessed by MCTS with respect to the number $N(s,a)$ of rollouts performed from (s,a), initialized to 0. MCTS maintains such a number for every applicable a in s. Lines 1 and 2 in MCTS selects a state and an applicable action with a trade off between less frequently updated and promising ones. Line 3 is an incremental update of the cost-to-go using Rollout, averaged over all rollouts performed in s and $\tilde{a}$. The successor s' of s returned by Sample$(s, \tilde{a})$ (in Rollout) is added to the envelope. The other states met along a rollout are *not* maintained in the envelope. The policy π_r used by Rollout can be $\mathrm{argmin}_a Q_0$ or any random policy. Within the enveloppe, π is maintained for newly expanded or updated states (lines 5). Ancestors of s in the envelope are updated bottom up until s_r; this can be performed by a procedure such as Algorithm 9.8. The *termination condition* stops MCTS when planning time is over or no open state remains in the envelope. At this stage, $\pi(s_r)$ is the best action in s_r estimated by MCTS.

The UCT algorithm. Algorithm UCT (for "Upper Confidence Trees") instantiate MCTS with a particular sampling strategy. It expands, to a bounded depth, a tree rooted at the current node. It develops this tree in a non-uniform way. At an interior node of the tree in a state s, it selects a *trial* action $\tilde{a}$ with the strategy described subsequently. It samples a successor s' of s along $\tilde{a}$. It estimates the value for s' (in Line 1) with a recursive call to UCT-rollout on s' with the cumulative cost of the rollout below s'. It uses this estimate to update (in Line 2) $Q(s, \tilde{a})$ by averaging over all previously sampled successors in $\gamma(s, \tilde{a})$ (as is done in SLATE or MCTS in Line 3).

```
UCT($s, d$)
    until $termination\ condition$ do
        UCT-rollout($s, d$)
UCT-Rollout($s, d$)
    if $s \in S_g$ then return 0
    if $d = 0$ then return $V_0(s)$
    if $s \notin Envelope$ then
        add $s$ to $Envelope$
        $N(s) \leftarrow 0$
        foreach $a \in Applicable(s)$ do
            $Q(s,a) \leftarrow 0;\ N(s,a) \leftarrow 0$
    $\tilde{a} \leftarrow$ Select($s$)                    // update with tradeoff in Equation 9.13
    $(s', c) \leftarrow$ Sample($s, \tilde{a}$)                       // $c = \mathrm{cost}(s, \tilde{a}, s')$
1   $cost\text{-}rollout \leftarrow c +$ UCT-rollout($s', d-1$)
2   $Q(s,\tilde{a}) \leftarrow [N(s,\tilde{a}) \times Q(s,\tilde{a}) + cost\text{-}rollout]/(1 + N(s,\tilde{a}))$
    $\pi(s) \leftarrow \mathrm{argmax}\{Q(s,a) \mid a \in Applicable(s)\}$
    $N(s) \leftarrow N(s) + 1$
    $N(s,\tilde{a}) \leftarrow N(s,\tilde{a}) + 1$
    return $cost\text{-}rollout$
```

Algorithm 9.27. UCT, a Monte-Carlo Tree Search procedure.

UCT is called repeatedly on a current state s until time runs out. When this happens, the solution policy in s is given by $\pi(s) = \mathrm{argmin}_a Q(s, a)$. This process is repeated on the state observed after performing the action $\pi(s)$. UCT can be stopped anytime.

The strategy for selecting trial actions is a trade-off between promising actions and those that need further exploration. Let us denote the untried actions in s at some stage as $Untried(s) = \{a \in Applicable(s) \mid N(s,a) = 0\}$. A trial action $\tilde{a}$ in a state s is selected as follows:

$$\mathsf{Select}(s) = \begin{cases} \text{arbitrary } a \in Untried(s) & \text{if } Untried(s) \neq \varnothing, \\ \text{argmin}_a\{Q(s,a) - C \times [\log(N(s))/N(s,a)]^{1/2}\} & \\ & \text{if not.} \end{cases} \quad (9.13)$$

where $N(s,a)$ is the number of time a has been sampled in s, $N(s)$ is the total number of samples in that state, and $C > 0$ is a constant. The constant C fixes the relative weight of *exploration* of less sampled actions (when C is high) to *exploitation* of promising actions (C low). Its empirical tuning significantly affects the performance of UCT. The choice in *Untried* when not empty can be heuristically guided.

One can prove that this selection strategy minimizes the number of times a sub-optimal action is sampled and that UCT converges asymptotically to the optimal solution.

All approaches described in this Section 9.5.4 can be implemented as memoryless procedures (in the sense discussed in Section 9.5.1). They are typically used in a receding horizon MDP-Lookahead schema. This simplifies the implementation of the planner, in particular when the lookahead bounds are not uniform and have to be adapted to the context. This has another important advantage in non-stationary domains. These procedures can generate non-stationary policies, possibly stochastic. Indeed, an actor may find it desirable to apply a different action on its second visit to s than on its first. For indefinite horizon problems in particular, non-stationary policies can be shown to outperform stationary ones.

9.6 Discussion and Bibliographic Notes

9.6.1 MDP Planning Algorithms and Heuristics

The Dynamic Programming foundations and main algorithms go back to the early work of Bellman, Putermann, Bertsekas, and Tsitsiklis [109, 919, 131] and other contributions discussed in the previous chapter. More recent studies revealed additional properties of the Value Iteration algorithm, e.g., complexity results with positive costs and lower bound heuristics [154], or sub-optimality bounds [467]. The propositions in Section 9.1 are demonstrated in [130]. Several extended and improved Value Iteration algorithms have been proposed, for example, with a prioritized control [40]; with a focus mechanism [349, 778, 1181]; with a backward order of updates from goals back along a greedy policy [268]; or with value estimation by random sampling in approximate value iteration [97, 1030].

Policy Search methods (not the be confused with Policy Iteration) deal with parametrized policies π_θ and perform a local search in the parameter space of θ (for example, gradient descent). The survey in [288] covers in particular their use for continuous space domains and reinforcement learning problems.

LAO* is developed in [469] as an extension of AO* [856]. The Find&Revise schema was proposed in [157], together with several instantiation of this schema into heuristic search algorithms such as HDP [157], LRTDP [158] and LDFS [160]. A few other heuristic algorithms are presented in their recent textbook [392, chap. 6 & 7]. RTDP has been introduced in [99]. The domain-configurable control technique presented in Section 9.3 was developed in [658].

The FF-Replan planner has been developed in [1208] in the context of the International Planning Competition. A critical analysis of its replanning technique appears in [723], together with a characterization of "probabilistically interesting problems." These problems have dead ends and safe solutions. To take the latter into account, an online receding horizon planner, called FF-Hindsight[1209] relies on estimates through averaging and sampling over possible determinizations with a fixed lookahead. The RFF algorithm has been proposed in [1087]; it has been generalized to hybrid MDPs with continuous state variables [1085].

Linear programming was introduced as a solution method for MDPs in [293] in the early 1960s. The book [34] provides a thorough account of linear programming and other methods for solving C-MDPs. The i-dual heuristic search algorithm originates from [1102], whereas the i2-dual algorithm and occupation measure heuristics were described in [1104].

Monte Carlo rollouts (named after the casinos of Monte Carlo) have been used very early in computational physics, e.g., quantum Monte Carlo methods and particle physics simulations. Particle filtering techniques adapt these approaches to signal processing [289]. Monte Carlo Tree Search techniques have been developed for game trees [182]. MCTS won the computer Go tournament in 2005 [260]. Further extensions and combination with neural networks lead to several developments in planning, scheduling and games, and the well-known success of AlphaGo and AlphaZero [1018, 1019].

The SLATE procedure is due to [595]. UCT was proposed in [623]. An AO* version of it is described in [162]. UCT was implemented into a few MDP planners such as PROST [597]. An extension of UCT addressing POMDPs is studied in [1015].

UCT converges on a indefinite horizon MDP: the probability of not finding the optimal action at the root node goes to zero at a polynomial rate as the number of rollouts grows to infinity (Theorem 6 in [623])

Several contributions have exploited determinization techniques in probabilistic planning, e.g., for pruning unnecessary Bellman update [169], for performing Graphplan like reachabilitiy analysis [583], or for computing heuristics for the mGPT planner [159]. Proposition 9.18 is demonstrated in the latter reference.

The regrouped operator-counting and the projection occupation-measure heuristics were introduced in [1104]. The authors conjectured that both heuristics are actually equivalent; was shown to be the case in [613]. The idea of deriving probabilistic planning heuristics from general projections (beyond single variables) was explored much later after the introduction of h^{pom} in [612]. This paper developed probabilistic pattern-database (PPDB) heuristics for MAX-PROB, where multiple projection heuristics are combined via multiplication in place of addition. PPDB heuristics for SSPs were introduced in [611]. Besides projections, literature has also studied more general types of probabilistic abstractions [614, 615].

For many planners, deep dead ends can lead to inefficiency or even to nontermination (for example, as in RTDP and LRTDP). Dead ends can be detected, but unreliably, through heuristics. They are safely avoided through the unbounded growth of the value function V with positive costs, as in Find&Revise instances and other variants, for example, [629], but this can be quite expensive. Allowing for real costs requires algorithms able to check and avoid dead ends, as in [1089], or in the GSSP model [631]. GSSP accounts for maximizing the probability of reaching the goal, which is an important criterion, also addressed by other means in [919] and [1088]. The approaches in [633] and [1086] for the S^3P model goes one step further with a dual optimization criterion combining a search for a minimal cost policy among policies with the maximum probability of reaching a goal. An explanation-based learning technique to acquire clauses that soundly characterizes dead ends is proposed in [630]. These clauses are easily detected when states are represented as conjunction of literals. They are found through a bottom-up greedy search and further tested to avoid false positives. This technique can be usefully integrated into the generalized Find&Revise schema proposed for the GSSP model [631].

9.6.2 Factored and Hierarchical MDPs

Dynamic programming techniques for MDPs with a structured or factored representation are studied in [171]. An elaborate and scalable techniques approximation for MDPs represented with DBNs is studied in [452], with a value function as a linear combination of basis functions for subsets of the state variables.

The PPDDL language [1212] is supported by planners such as mGPT [159] or PFD [1056]. RDDL is partially supported by a few planners, e.g., GLUTTON [632], BEAVER [928], SPUDD and PROST [598]. Their respective merits in various benchmarks are analyzed in [937].

Symbolic techniques with binary and algebraic decision diagrams have also been used in probabilistic planning, e.g., a symbolic Value Iteration in the SPUDD planner [505]. These techniques are used in an RDTP algorithm [344], or a symbolic LAO* [343]. The nondeterministic MBP planner have been extended to MDPs [768].

Several algorithms have been proposed to take advantage of the structure of a probabilistic planning problem. This is the case, for example, for hierarchical MDPs of the HiAO* algorithm [786]. Different methods can be used to hierarchize a domain, e.g., [426]. Model minimization techniques have been studied in [284]. A kernel decomposition approach has been developed in [281]. Approximate solutions to large MDPs with macro actions, that is, local policies defined in particular regions of the state space are studied in [485]. The DetH* algorithm [90] clusters a state space into aggregates of closely connected states, then it uses a combination of determinization at the higher level and Value Iteration at the lower level of a hierarchical MDP.

Sparse probabilistic domains have been studied in e.g., [192, 717]. The path compression technique of Algorithm 9.20 is detailed in the latter reference.

9.6.3 Continuous and Partially Observable MDPs

MDPs in continuous state and action spaces use generally a flat representation: $S \subseteq \mathbb{R}^n$ and $A \subseteq \mathbb{R}^m$. States and actions as vectors of real numbers, bounded in appropriate intervals, are quite popular for modeling robotics and control problems, e.g., [685, Sec.8.5.2]. When actuation is performed at discrete time points (e.g., at a fixed frequency), we are still in the framework of discrete transition systems. This continuous MDP model is equivalent to having a single parametrized action and the choice of the parameter values to apply to the current actuation point.

Bellman equation is easily extended with probability density functions to continuous MDPs for the bounded and the discounted infinite horizon cases. The planning problem can be addressed by computing the value function at hyper-rectangles in S, the boundaries of which are defined by lower and upper bounds on the values of each state variable. This is called the *Rectangular Piecewise Constant* (RPWC) representation of V [345]. RPWC is consistent with Bellman updates (with a caution for adaptive discretization), allowing for algorithms similar to Value Iteration. RPWC can be used to discretize the density functions instead of V, with similar properties [708]. When the cost function is linear, a *Rectangular Piecewise Linear* (RPWL) discretization of V is also amenable to Bellman updates with better results [729].

Monte Carlo Tree Search methods are a powerful alternative to Dynamic Programming approaches for continuous MDPs. The so-called *Action Progressive Widening* adapt the UCT sampling strategy to a continuous space [218]. An interesting variant systematically selects the minimum, median and maximum points of the multi-dimensional action space before pursuing with random samples [259, 137].

Hybrid MDPs have continuous and discrete state variables. They have been addressed with extensions of the previous approaches, with various Linear Programming techniques [484, 662], and with extensions of heuristics search methods, such as Hybrid AO* [788].

Other types of models are needed when actions are continuous functions of time. For example, Time-dependent MDPs handle time continuous actions but assume discrete transition probability distributions; they can be addressed with Dynamic Programming methods [172]. The model of Generalized Semi-Markov Decision Processes does not require discretization [1216]. It can handle uncertain action durations, but it does not manage plan duration, an issue partly addressed with exponential distributions in [753].

Partially Observable MDPs (introduced in Section 8.5.3) are MDPs in the belief state, which is continuous. Many of the mentioned continuous MDP approaches have been applied to POMDPs, e.g., the RPWC and RPWL approximations. When S and A are discrete, POMDPs with a discretized belief space draw much attention and concerns for addressing its exponential size in $|S|$, which itself is exponential. Dynamic Programming and Heuristic Search methods to discretized POMDPs have been proposed [570, 1041]. Approximate methods that focus Bellman updates on a few belief points (called *point-based methods*) are surveyed in [998]; they are compared to an extension of RTDP [161]. Parametrized POMDPs have been addressed with policy search techniques [846].

Monte Carlo methods, which conveniently sample continuous domains, scale up better than previous approaches. For example, the POMCP planner as been applied to large POMDPs [1015]. The DESPOT planner combines sampling and anytime search [1045]. The progressive widening techniques have been extended to POMDPs[1065, 719]. Online algorithms for POMDPs are surveyed in [961]. We already mentioned the termination problem for goals expressed in the belief space. Fortunately this problem that can be addressed with termination actions [466], in particular with Monte Carlo methods.

Robotics offers many POMDP use cases [897, 367, 454]. However applications often require a more flexible hybrid model, with observable state variables, as well as non-observable ones. The latter are estimated, as in POMDPs, from indirect observation variables. This model is called Mixed Observability MDPs (MOMDPs) [858, 48]. It is attracting interest for, e.g., target tracking [279], navigation [290], or conservation and natural resource management applications [888].

9.7 Exercises

9.1. Run Policy Iteration on the problem in Figure 9.10, starting with the policy $\pi = \{(s_0, \mathsf{a}), (s_1, \mathsf{c}), (s_2, \mathsf{d}), (s_3, \mathsf{f})\}$.

- Compute $V^{\pi_0}(s)$ for the four non-goal states.
- What is the greedy policy of V^{π_0}?
- Iterate on the above two steps until reaching a fixed point.

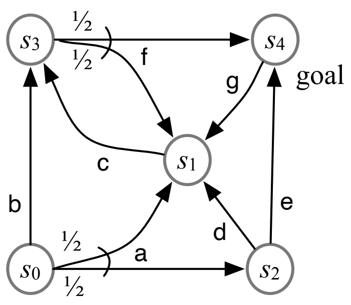


Figure 9.10. A simple SSP with unit cost actions.

9.2. Repeat Exercise 9.1 on the problem in Figure 9.11, starting from the following policy: $\pi_0(s_1) = \pi_0(s_2) = \mathsf{a}, \pi_0(s_3) = \mathsf{b}, \pi_0(s_4) = \mathsf{c}$.

9.3. Run Value Iteration on the problem in Figure 9.10, with $\eta = 0.1$. Assume that the **foreach** statement iterates through states in order of increasing subscripts, and the argmin operator breaks ties by choosing the action that comes first alphabetically. The heuristic function is $V_0(s_i) = \begin{cases} 0, & \text{if } s_i \text{ is a goal state,} \\ i, & \text{otherwise.} \end{cases}$

9.4. Repeat Exercise 9.3 on the problem in Figure 9.11 with $\eta = .5$, with the following two heuristic functions:

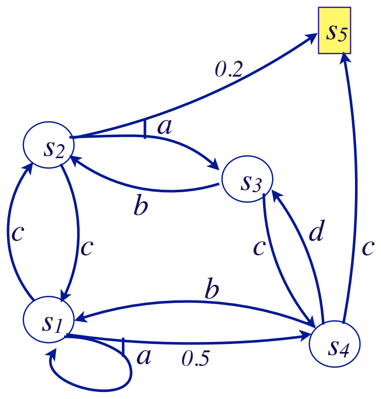


Figure 9.11. An SSP problem with five states and four actions a, b, c, and d; only action a is nondeterministic, with the probabilities shown in the figure; the cost of a and b is 1, the cost of c and d is 100; the initial state is s_1; the goal is s_5.

- $V_0(s) = 0$ in every state.
- $V_0(s_1) = V_0(s_2) = 1$ and $V_0(s) = 100$ for the two other states.

9.5. In the problem of Figure 9.11, add a self loop as a nondeterministic effect for actions b, c, and d; that is, add s in $\gamma(s, a)$ for these three actions wherever applicable. Assume that all the distributions are uniform. Solve the two previous exercises on this modified problem.

9.6. Implement and run algorithm Value Iteration for a few problem instances of the domain PAM$_p$ (Example 8.9). Up to how many containers does your implementation scales up?

9.7. Run AO* on the domain of Figure 9.4 with the heuristics V_1 of Section 9.3.

9.8. Modify the domain of Figure 9.4 by making the state s_{12} an immediate dead end instead of a goal; run AO* with the heuristics V_0 and V_1 of Section 9.3.

9.9. Run LAO* on the problem in Figure 9.10, with the same η and V_0 as in Exercise 9.3. Assume that in the select statement, the tie-breaking rule is to select the state s_i for which i is smallest.

9.10. Prove that algorithm LAO* is an instance of the Find&Revise schema.

9.11. Modify the domain of Figure 9.4 by changing $\gamma(s_9, \text{a}) = \{s_3, s_8\}$ and making the state s_{15} an immediate dead end instead of a goal. Run LAO* and ILAO* on this problem and compare their computation steps.

9.12. Run RFF on the problem in Figure 9.10 with $\theta = 0.7$, using a Forward-Search algorithm that always returns a least-cost path to a goal state. Give the following:

(a) Each possible history and its probability. If there are more than four histories, then say so and give the first four of them.
(b) The probability that the actor will reach the goal.

9.13. Run RFF on the problem of Figure 9.11 with $\theta = 0.7$. Suppose the Det-Plan subroutine calls the same Forward-Search algorithm as in the previous exercise, and turns the plan into a policy. What is π after one iteration of the "while" loop?

9.14. Prove that algorithm RFF is complete when using a complete Det-Plan deterministic planner.

9.15. Run Algorithm 9.20 on the problem of Figure 9.11; compare with the computations of RFF on the same problem.

9.16. Specify the SLATE procedure (Algorithm 9.25) as an anytime algorithm implementing an incremental backup at each increase of the depth d. Implement and test on a few domains.

10 Reinforcement Learning

Reinforcement learning (RL) is about learning to act with probabilistic models. RL extends homeostasis regulation to complex behaviors. It parallels metaphorically the adaptation mechanisms of natural beings to their environment, with on feedback sensing and means for evaluating what's good and what's not. Adaptation is a key feature of intelligence: an autonomous actor should be able to learn from its actions. With continual learning, an actor can cope with a continually changing environment.

In the following, we first introduces the main principles and terminology of reinforcement learning. Section 10.2 presents a simple form of Q-learning, a generic value-based RL algorithm. Section 10.3 addresses how to generalize a learned relation with a parametric representation. We then introduce neural network methods, which play a major in learning and are needed for the remaining sections, about deep RL (Section 10.5), and policy-based RL (Section 10.6). The issues of aided reinforcement learning with shaped rewards, imitation learning and inverse reinforcement learning are addressed next. Section 10.8 is about probabilistic planning and RL. A discussion, bibliographical notes, and exercises end this chapter. Appendix B recaps the mathematical notations used.

10.1 Principles of RL

Reinforcement Learning (RL) *interleaves acting and learning*, possibly in a *continual learning* framework, to improve an actor's performance for a given task or goal by a trial and error interactions with the world. RL may or may not have a domain model. It relies on a *reward function*, which defines, in an indirect way, the actor's purpose, i.e., *what* it has to do. RL is used to find out *how* to do it. This reward function is assumed to reflect a reliable and robust specification of the task to learn. This is an important assumption that needs to be kept in mind.

The RL learner tries to remember from its past activity which actions in which states led to higher rewards and which were bad, and, possibly, *generalize* this knowledge in order to use it for taking good actions in the future. It seeks to learn how to act by maximizing the long-term perceived benefit of its actions.

A learner on its own may not have a teacher. It relies solely on a feedback from the environment following its actions. This feedback is a number: the *reward* for going from a state s to s' with a performed action a. RL is an *active learning* mechanism: the learner acts not only to achieve its task, but also to learn more about how best to achieve it. For example, suppose our learner is a cook whose task is to cook a paella; her reward is the number of likes from her guests. Should she stick to the recipe she learned or should she try possible variants to improve her cooking? Such a learner has to solve a tradeoff between *exploitation* and *exploration*: whether to stay

Free pre-publication, for personal use only. To be published by Cambridge University Press.

on a safe, well-known track or to take the risks and efforts to explore possibly better unknown ones. Exploitation makes the best with what is already learned to maximize the behavior benefits. To learn the best behavior, exploration has to try options that are not known enough.

In general it is not feasible to try every possible action in every possible state. *Generalization* is a major issue for RL, as well as for other learning approaches. It is about extending what has been learned in one situation to 'similar' situations. The holy grail is to learn much from a few trials. A further generalization ambition is to *transfer* what has been learned for a given task to other 'similar' tasks.

The trial and error approach may entail a high risk, unacceptable in critical applications. With a domain simulator, part of the risk can be avoided: actions are simulated before being performed in the real world. Current RL techniques remain demanding, in computational as well as in *sampling complexity*, i.e., in the number of trials required to learn a good policy. For that also, simulation is needed.

A learner may get help from a teacher, e.g., with demonstrations of good behaviors in certain situations, or with advice about how to choose actions. Instead of (or in addition to) rewards, the learner gets trajectories from a teacher's demonstrations. A pedagogical teacher may also *shape* the rewards such as to ease the learning.[1] It may organize the learner's tasks into a teaching *curricula*. But even with a teacher, the capability to generalize what has been learned remains essential.

The usual and convenient framework of RL is the probabilistic MDP representation. Actions have probabilistic effects. The actor seeks to learn a policy which maximizes the total expected reward. Many RL approaches consider *stochastic policies*, which map states to probability distributions on actions. We focus here on deterministic policies, which are simpler and easier to learn.

In a process maintenance MDP, the planner seeks a policy that optimizes the actor's behavior over a bounded or an infinite horizon (see Section 8.3.1). The learner seeks to maximize the average reward per step or the total *discounted* reward over its lifetime. In goal reachability or episodic tasks MDPs (i.e., SSPs), the planner searches an optimal the policy that reaches the goal or performs the task. The learner seeks to achieve the task to be learned while maximizing its total reward. The task is expressed through a reward function. A learning *episode* is a trial for achieving the task. It involves a *finite* number of steps and necessarily terminates when the task succeeds or fails, on reaching a terminal state or a termination action.

This chapter is focused on *indefinite horizon episodic* RL, where learning seeks to maximize the expected total reward for an episode. Hence we avoid the drawbacks of discount factors (see Section 8.3.2). This formulation of RL for learning a task with its associated reward function extends naturally to a goal-conditioned formulation, for learning a policy for a given goal or set of goals (as per Definition 8.3).

RL in the MDP frameworks comes in many flavors, among which the following:

- *Model-based* vs *model-free* RL. In the latter, the learner does not have and does not use the transition function γ nor the corresponding probability distributions; it acts and observes its states and rewards. In model-based RL, an approximate

[1] Reward shaping is a wide spread practice, from animal trainers to school teachers and coaches.

world model is learned from acting, used as a proxy of the world by planning actions good for that model, then improved by further acting. This progressively learned model is used in the exploitation, and possibly the exploration stages.

- *Value-based* vs *policy-based* model-free RL. The latter estimates an optimal a policy π^* given its past experiences. The former estimate V^* or Q^*, the optimal value or action-value functions. This is in a way similar to Value Iteration *vs* Policy Iteration algorithms.
- *On-policy* vs *off-policy* RL. The former interleaves improving a policy and using that same policy to act. The latter acts according to a policy different from the one it is trying to optimize, which can be convenient for exploration.

We mostly focus here on model-free RL, with value-based methods (sections 10.2 to 10.5), and policy-based ones (Section 10.6).

10.2 Tabular Value-Based RL

In value-based RL the actor seeks to estimate from trial and errors, V^* or Q^*, from which it easily derives a policy. This might be done in a *batch mode* from all experiences, accumulated in a first costly stage. Preferably, learning can progress incrementally, each new experience improves the current best value estimates, and increases the quality of the actor's behavior. In this mode, acting serves the task achievement purpose as well as the learning purpose. We focus here on incremental RL. At each stage, the learner progresses by updating its estimates with respect to the difference of their values from t to $t + 1$. Because of these incremental updates, the corresponding techniques are called *temporal difference* or TD learning methods, exemplified with the Q-learning algorithm presented here. We first introduce the main concepts in a very simple case. We then present Q-learning for a *tabular* representation: the values learned incrementally are simply cached in memory as a table and used as is. Other value-based algorithms and extension to a structured representation are then discussed.

10.2.1 A First Intuition

To give a first idea about Q-learning and its main ingredients, let us warm up with a very simple case. Consider an actor that has n actions, anyone of them can be used alone to perform the task at hand, with differing results. For this elementary actor, each trial of an action is independent of the previous and following trials. The only information available to the actor is a varying reward associated with performing an action. The actor wants to learn what's its best action in average for the task.

Example 10.1. Eva is a cook that knows three possible recipes a, a', a'' for making a paella. She wants to know what is the best recipe giving the observed average satisfaction of her fixed set of guests after several trials.[2] After one trial of each of

[2]Eva may have a more demanding objective, such as maximizing the total expected satisfaction of her guests over many meals. In this case Eva may end up with a strategy switching recipes. This interesting case, related to the so-called *n-arms bandit problem*, will be discussed in Section 10.9.

the three recipes, the guests rank a, a', a'' respectively 7.2, 5.4, and 6.8, in a scale $[0, 10]$. These ranks are the rewards Eva receives. Another trial of a' is ranked 8; a' moves up to an estimated “quality” of 1/2(5.4 + 8)=6.7. A following trial of a' is ranked 7.6, giving a current quality of 1/3(5.4 + 8+7.6)=7. This goes on until Eva is confident about the evaluation of her three recipes. □

Let $r(a,i) \in \mathbb{R}$ be the reward received after running action a the i^{th} time. The actor estimates the value of an action a that has been performed k times by its average reward:

$$\begin{aligned} Q_k(a) &= \frac{1}{k}\sum_{i=1}^{k} r(a,i) \\ &= \frac{1}{k} r(a,k) + \frac{k-1}{k} Q_{k-1}(a) \end{aligned}$$

By dropping in the above formula the index of the current trial we get an incremental *update rule* of $Q(a)$ with respect to the last observed reward $r(a)$:

$$Q(a) \leftarrow \alpha r(a) + (1-\alpha) Q(a). \tag{10.1}$$

The parameter $\alpha = \frac{1}{k}$ is called the *learning rate*. There can be different learning rates for different actions, but one seek to try every action as often. When $\forall a, k \longrightarrow \infty$, the choice of the action which maximizes the average reward is given by $\mathrm{argmax}_a\{Q(a)\}$. As long as the exploration of alternative actions has not been sufficient, the actor needs to try other options according to some heuristics. These can be expressed as a procedure Select, defined for example as:

$$\mathsf{Select} = \begin{cases} \mathrm{argmax}_a\{Q(a)\} & \text{with probability } (1-\epsilon) \\ \text{random } a' \neq \mathrm{argmax}_a\{Q(a)\} & \text{with probability } \epsilon, \end{cases} \tag{10.2}$$

where ϵ is decreasing with experience. Such a strategy is called ϵ*−greedy*; it is most often greedy in selecting the currently best option, except for a few exploration cases. Alternatively, Select may choose an action according to a probabilistic sampling distribution, e.g., the Boltzmann sampling, according to a probability distribution proportional to $e^{-Q(a)/\tau}$, where τ is decreasing with experience.

The simple update rule 10.1 appears in different forms and plays an important role in reinforcement learning. It is referred to as the *temporal difference* update rule (the current value of Q affects the learned next value). It leads to a family of temporal difference RL approaches. The *learning rate* α becomes too small for large k. When the environment is stationary, the update of $Q(a)$ with 10.1 becomes increasingly weak. If the environment is not stationary, one may keep $\alpha < 1$ constant.

The update rule 10.1 starts with some initial value $Q(a) = q_0$ when a has never been tried. A high value for q_0 can be a bonus for the exploration of new actions.

This basic case is too simple. It does not relate the quality of an action $Q(a)$ to the state in which a is performed; nor does it consider the composition of several actions for a task. In Example 10.1 a paella recipe is made of a dozen of actions all relevant for the final quality of the dish. A given action may have no immediate reward but a significant influence in the final reward. However, this simplistic example helps introducing the main notions of Q-learning, developed next.

10.2.2 Q-learning for a Tabular Representation

Let us now consider the case where several interdependent nondeterministic actions are needed to perform a task. In a standard reinforcement learning formulation, an actor wants to learn a policy for performing the task at hand on the basis of rewards from trial and error. The framework is that of MDP (Chapter 8). We want to synthesize a solution policy to a goal-directed MDP problem (see Definition 8.3), where the goal is to terminate the task. But we do not know the domain model. The actor learns by repeatedly trying to achieve its task. Each trial terminates with success or failure after a *finite* number steps. In each, an action a applicable in current state s is performed; the next state s' and a reward $r(s, a, s') \in \mathbb{R}$ are observed.

In the planning chapter we sought to minimize the expected sum of the cost of the actions obtained by following a solution policy π from a state s to a goal. Here we seek to maximize the expected sum of rewards. Earlier we defined a value function V with respect to action costs (Equation 8.3). We can define it analogously for rewards:

$$V^{\pi}(s) = \sum_{s' \in \gamma(s,\pi(s))} \Pr(s'|s, \pi(s))[r(s, \pi(s), s') + V^{\pi}(s')]$$

Similarly for Q, called here to the *action-value* function:[3]

$$Q^{\pi}(s, a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s, a)[r(s, a, s') + V^{\pi}(s')]$$

The Bellman equation for Q (Equation 9.4) can be restated in this context as:

$$Q(s, a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s, a)[r(s, a, s') + \max_{a'}\{Q(s', a')\}] \qquad (10.3)$$

Without prior knowledge of the domain model, i.e., without γ, we cannot solve Equation 10.3 with the algorithms seen earlier. Instead of that, we can estimate $Q(s, a)$ from the statistics of trial and error with incremental updates, as we did in the elementary case. However, the simple update rule 10.1 does not integrate in Q the effect of a follow-up action a'. Here, the update on $Q(s, a)$ has to take into account the action-value of the following stage to maximize the total expected reward. This is done, as in Bellman Equation 10.3, with the following update rule, at the core of the Q-learning algorithm:

$$Q(s, a) \leftarrow \alpha[r(s, a, s') + \max_{a'}\{Q(s', a')\}] + (1 - \alpha)Q(s, a) \qquad (10.4)$$

[3]In Chapter 9, Q was called the *cost-to-go*. Here it is called the *action-value* function, since we no longer have costs but rewards, and we move from minimization to maximization.

In the right hand side of Equation 10.4, $Q(s,a)$ is the old value of Q, $\max_{a'}\{Q(s',a')\}$ refers to the action to be executed next, the update term as per Bellman equation is $r(s,a,s') + \max_{a'}\{Q(s',a')\}$. The current gap in the estimated value is the difference $[r(s,a,s') + \max_{a'}\{Q(s',a')\} - Q(s,a)]$.

Q-learning proceeds by repeatedly trying to perform the task a given number of time. In each trial, called an *episode*, the algorithm starts from some initial state, randomly drawn in a given set in S_0 of possible initial states. An episode terminates after a *finite* number of steps, when the task succeeds, fails, or some bound is reached. In each step, a selected action a is performed, the next state s' and a reward $r(s,a,s') \in \mathbb{R}$ are observed. The algorithm is off-policy: the selected action (line 1) is not necessarily the current policy given by $\text{argmax}_a Q_\theta$. The online version given in Q-learning can be transformed into a batch version of Q-learning which learns from a history of a recorded number of episodes obtained with some exploration strategy.

```
Q-learning
    initialize $Q$
    for each episode do
        randomly draw a starting state $s$ from $S_0$
        until episode termination do
1           $a \leftarrow$ Select($s$)                    // selects $a \in Applicable(s)$
2           perform action $a$
3           observe resulting state $s'$ and reward $r(s,a,s')$
4           $Q(s,a) \leftarrow \alpha[r(s,a,s') + \max_{a'}\{Q(s',a')\}] + (1-\alpha)Q(s,a)$
5           $s \leftarrow s'$
```

Algorithm 10.1. Q-learning algorithm

Q-learning implements Q as lookup table over $S \times A$. It uses a function Select(s), similar to the elementary case, which favors the action $\text{argmax}_a\{Q(s,a)\}$ among applicable actions, while allowing for the exploration of other actions with a frequency decreasing with experience, controlled with the ϵ parameter. Large ϵ favors exploration. The learning rate parameter $\alpha \in [0,1]$ is set empirically. When α is close to 1, Q follows the last observed rewards by weighting down previous experience of a in s; when α is close to zero, the previous experience is more important and Q changes marginally. If the environment is stationary (i.e., when the distributions remain invariant over time), α can be set decreasing with the number of instances (s,a) encountered. Here also, the initial values of $Q(s,a)$ may favor exploration.

The main features of Q-learning are the following:

- it is model-free,
- it learns a policy (implicit in the pseudo-code) as $\pi(s) = \max_a Q(s,a)$,
- it is off-policy: it acts with Select(s) which may be different from learned policy allowing for exploration,
- its update rule performs a local step at (s,a) of policy evaluation and improvement.

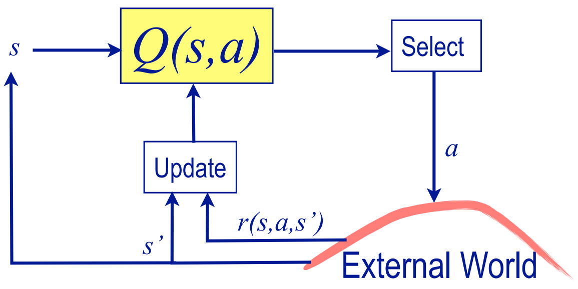


Figure 10.1. A schematic view of Q-learning with a tabular value function Q.

An episode terminates when the task succeeds, fails or a bound on the number of steps is reached. The algorithm may be run with a fixed number of episodes, or stoped when exploration is sufficient, i.e., when enough statistics have been gathered for every pair (s, a), allowing the learner to act with a policy $\pi(s) = \max_a Q(s, a)$. One can prove the asymptotic convergence of Q-learning to optimal policies when each pair (s, a) has been met infinitely often. Intuitively, the arguments for this convergence combines those for the convergence of the simple update rule 10.1 to the true vales and the Bellmann equation to the optimum.

But the convergence of the simple Q-learning is too slow for most practical applications, ruling out learning by performing actions in the real world. If learning can start using a *simulator*, the actor might gather offline as much initial statistics as possible in simulation. After that, it would get into a *continual learning and acting* stage: when it needs to perform that same task, it would simply run lines 1 through 5 of Q-learning and possibly decrease α for future updates.

This algorithm is referred to as Q-learning for a *tabular* representation: it assumes that Q is maintained in memory as a global data structure, i.e., a 2D table over all (s, a) pairs, and that S and A are sets of ground states and actions. We'll see in Section 10.3 how to overcome the limitations of the memory-based approach. There are several variants of the memory-based Q-learning algorithm such as SARSA and DYNA algorithms that will be discussed in Section 10.9.

RL with a simulator. Q-learning explores with an open "curiosity" possibly dangerous transitions that may have high negative rewards. It has no means for avoiding dead-ends. Learning with trial and error entails a risk incompatible with critical applications, for which extensive modeling and testing are usually required. One does not learn to build bridges through trial and error, but a bridge collapse is taken as an unfortunate learning event to improve models.

A simulator reflects a model of domain, but there are differences between a model implemented in a simulator and the one implicitly or explicitly learned by RL. To put it briefly, the former is about *know-what* to do, while the latter is about *know-how* to do things. A simulator model is a low level description of what might possibly happen in a domain in response to some elementary actions. For example, in board games, a simulator gives trivially the board state after a move. In robotics, a simulator implements the robot and the environment geometry, kinematics, dynamics as well as the physics of actuation and sensing to compute state updates after a robot command or an event (see Part VII). The models of interest at the RL level are about how best to act in order to win the game, or to perform a robotics task. None of that is available in the simulator model.

As an analogy, consider the difference between a model of the ballistic of a thrown dart and a model for dart playing. For a human player, the former remains often intuitive, while the latter requires significant training. A robot with a simulator of a dart dynamics would learn much faster how to play darts.

In a realistic domain, a simulator is necessary to a learner for convergence at a practical cost. But a simulator may be wrong and ignore possibly relevant variables. A good simulator is mandatory for critical domains with dead ends or possibly dangerous actions, e.g., a robot should not get into a slippery zone near a cliff.

One way of interfacing Q-learning with a simulator is to use two stages. In a first stage the learner runs a variant of Q-learning where lines 2 and 3 are replaced with:

$$(s', r(s, a, s')) \leftarrow \mathsf{Simulate}(a, s)$$

where Simulate calls the simulator and returns the next state and the reward. When sufficient simulated exploration has been performed, the learner starts in the second stage acting and learning in the real world. To account for the limitations of the simulation models, the learner improves its knowledge starting with a higher α. At this second stage, an ϵ-greedy Select may rely on $\mathrm{argmax}_a\{Q(a)\}$ with some confidence. However, it should not propose a different action without performing a look-ahead with Monte Carlo rollout outs in order to avoid dubious explorations and remain safe.

10.3 Parametric Value-Based RL

The Q-learning algorithm and other variants, as presented in the previous section, have a huge drawback: they memorize the learned Q as lookup table and require the knowledge of $Q(s, a)$ for every (state, action) pair. Except for trivial cases, this simply does not work. This is because of the size of the corresponding state and action spaces (e.g., Example 8.9). More importantly, this is also because the learner is unable to generalize from what it observes to unobserved cases: each experience gives it only a single point in a huge space. Evidently, this drawback is not specific to Q-learning, but holds for every memory-based representation.

10.3.1 Parametric *vs* Memory-Based Representations

Consider the simple case of a learner who observes a collection $\mathcal{D}$ of matching pairs of points in some spaces X and Y: $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid x^{(i)} \in X, y^{(i)} \in Y, 1 \leq i \leq N\}$. $\mathcal{D}$ is the learner's *training database*. The learner assumes that its observations are independent and identically distributed (the *i.i.d* assumption) according to a relationship to be learned and from which the learner would like to predict the value y matching some new given x. If the learner can only rely on its memory of the matching pairs in $\mathcal{D}$, then it will be at loss for finding y, unless by chance $x = x^{(i)}$, for some already experienced pairs in $\mathcal{D}$, a highly improbable situation when $\mathcal{D}$ is small compared to the dimension of X and Y.

Now, assume that X and (possibly) Y are metric spaces, endowed with a distance δ (see Section B.1). The learner can find the closest $x^{(i)}$ to x, according to δ, and estimate y with respect to the corresponding $y^{(i)}$. The nearest neighbor methods and similar non parametric classification techniques rely on exactly this idea.[4]

If X and Y are metric spaces, the learner might as well approximate the matching relationship exemplified in $\mathcal{D}$ with an easily computable parametric function $y = f_\theta(x)$, for a vector of parameters $\boldsymbol{\theta} \in \mathbb{R}^m$. A parametric function is a family of functions varying with some parameters.[5] One seeks the best parameter values for some specific use. Given a value for θ, f_θ will immediately provide an estimate of y for a new x.

In the simple case where X and Y are real numbers, a very well-known and widely used instance of this approach is *linear regression* or line fitting. Linear regression seeks an approximation with a straight line $f_\theta(x) = \theta_0 + \theta_1 x$. For each point (x, y) in $\mathcal{D}$, this approximation will predict $\tilde{y} = \theta_0 + \theta_1 x$ instead of y. A good approximation would set the vector $\boldsymbol{\theta} = [\theta_0, \theta_1]^\top$ such as to minimize some distance between the predicted $\tilde{y}$ and the targeted y over all pairs in $\mathcal{D}$. This can be expressed as minimizing the squared error *empirical loss function* for f_θ:[6]

$$Loss(f_\theta) = \sum_{(x,y)\in\mathcal{D}} (f_\theta(x) - y)^2 = \sum_{(x,y)\in\mathcal{D}} (\theta_0 + \theta_1 x - y)^2$$

In the case of linear regression, $Loss(f_\theta)$ is a convex function of θ_0 and θ_1. It has a single minimum, reached when its partial derivatives $\frac{\partial Loss(f_\theta)}{\partial \theta_0}$ and $\frac{\partial Loss(f_\theta)}{\partial \theta_1}$ are null:

$$\sum_{(x,y)\in\mathcal{D}} (f_\theta(x) - y) = 0, \text{ and } \sum_{(x,y)\in\mathcal{D}} (f_\theta(x) - y)x = 0 \tag{10.5}$$

These two equations are easily solved analytically, giving the optimal values of the two parameters in a closed form.

Example 10.2. Consider the cook Eva of Example 10.1 who discovers the booknote of a renown chef from which she wants to learn the "ideal" proportions of rice versus fish and seafood to put in a paella recipe. The booknote tells about 3 experiences. In the first one, the chef had $150g$ of fish and seafood for which he put $300g$ of rice. In the following experiences, the proportion where 300 vs 400, then 450 vs 700.

Giving this set $\mathcal{D} = \{(150, 300), (300, 400), (450, 700)\}$, Eva decides to model the ideal proportions with a simple linear regression. She solves Equation 10.5 for the estimated amount of rice given available fish and seafood as: $\tilde{y} = 1.33x + 66.6$. With

[4] In classification and clustering problems, Y may not need to be a metric space.

[5] E.g., the families of polynomials $\theta_0 + \theta_1 x + \ldots \theta_n x^n$ or gaussians $e^{\theta_0 + \theta_1 x + \theta_2 x^2}$ are parametric functions.

[6] *Loss* is the square of the Euclidean distance

this relation, Eva generalizes $\mathcal{D}$ for any amount of x. She can also use it to easily find how much rice, fish and seafood she would need for a paella of say 1500g.

Now the chef notebook may detail the amount of each of 12 ingredients in a paella, e.g., white fish, mussels, shrimps, seashells, onion, tomato, peas, pepper, garlic, olive oil, safran, sell and rice. Eva needs a model where f_θ is a function of 12 variables. □

When the space is $X = \mathbb{R}^n$, the above approach is called the *multivariable linear regression*. A point $\mathbf{x} = [x_1, \ldots, x_n]^\top \in X$ is a column vector; similarly for the parameters $\boldsymbol{\theta} = [\theta_0, \ldots, \theta_n]^\top$.[7] The linear approximation function is:

$$f_\theta(\mathbf{x}) = \theta_0 + \sum_{1 \le j \le n} \theta_j x_j, \quad \text{written as: } f_\theta(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x} \tag{10.6}$$

In the convenient dot product notation, the bias term θ_0 corresponds to an augmented $\mathbf{x}$ with a dummy constant element $x_0 = 1$. The empirical loss is:

$$Loss(f_\theta) = \sum_{(\mathbf{x},y)\in\mathcal{D}} (\boldsymbol{\theta} \cdot \mathbf{x} - y)^2$$

A slightly more general formulation is needed when $\mathbf{y}$ is also a vector, but f_θ remains a linear approximation. In that case $Loss(f_\theta)$ is still a convex function. The optimal values of the parameters $\boldsymbol{\theta}^* = \mathrm{argmin}_\theta\{Loss(f_\theta)\}$ can also be derived analytically from the partial derivates of $Loss(f_\theta)$.

When f_θ is not a linear function of $\boldsymbol{\theta}$, finding analytically the optimal parameters is more complex and often not feasible. A general approach for computing numerically the values of the parameters giving the minimal *Loss* is the gradient descent algorithm.

10.3.2 Gradient Descent

The idea is to search in the continuous parameter space for the minimum $\boldsymbol{\theta}^* = \mathrm{argmin}_\theta\{Loss(f_\theta)\}$ by following the direction given by the gradient of the loss function we try to minimize. This direction is the gradient vector

$$\nabla Loss(f_\theta) = [\partial Loss(f_\theta)/\partial\theta_1, \ldots, \partial Loss(f_\theta)/\partial\theta_n]^\top.$$

This is done with a sequence of update steps. Each update changes the parameters towards a direction decreasing the loss, i.e., with an update rule of the form $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla Loss(f_\theta)$.

Algorithm 10.2 is a simple instance of this general idea in the multivariable linear regression case. Its argument is a collection of observed pairs $(\mathbf{x}^{(i)}, y^{(i)})$, which is a subset of $\mathcal{D}$. In each iteration of the main loop, it makes one update step on each parameter θ_j following the direction given by the partial derivative at this local point. The update rule comes from $\frac{\partial Loss(f_\theta)}{\partial\theta_j}$, the *Loss* being over all observed pairs $(\mathbf{x}^{(i)}, y^{(i)})$ (integrating the constant factor from the partial derivatives in α). On reaching in some θ_j the minimum (which is global since in the linear case *Loss* is convex), the partial derivative is null. This θ_j stays at a fixed point. The convergence

[7]The transpose $\top$ of a row vector is a column vector (see Section B.2).

```
GradientDescent($\{(\mathbf{x}^{(i)}, y^{(i)}) \mid 1 \le i \le N\}$)
    until convergence do
        foreach $0 \le j \le n$ do
            $\theta_j \leftarrow \theta_j - \alpha \sum_{1\le i\le N}(\boldsymbol{\theta} \cdot \mathbf{x}^{(i)} - y^{(i)})x_j^{(i)}$        // Update rule
```

Algorithm 10.2. Pseudo code of a simple gradient descent algorithm.

criteria estimates how close to this fixed point for all j the gradient descent is. The learning rate α can be a fixed constant, but not too large to avoid over shooting. It can be set to a decreasing rate, together with the decreasing value of the gradient of *Loss*, until convergence.

When f_θ is linear, GradientDescent is guarantee to converge to the optimal parameters $\boldsymbol{\theta}^* = \text{argmin}_\theta\{Loss(f_\theta)\}$ with and appropriately chosen decreasing learning rate. In many cases, unfortunately, the matching relationship in $\mathcal{D}$ is not linear and cannot be approximated with a linear function. Consider for example the relations *(age, weight)* of a person, or *(age, reliability)* of an equipment. These are an increasing, then stable then decreasing functions, for which linear approximation are inadequate. Fortunately, the parametric approach, as just outlined, remains feasible with any family of functions, e.g., the polynomials or other exotic functions. As long as f_θ is *differentiable*, Algorithm 10.2 is adequate for finding its optimal parameters. However, for a nonlinear approximation function $Loss(f_\theta)$ is no longer convex. The algorithm can get stuck in local minima, requiring additional techniques for reaching a good approximation function (see Section 10.9).

GradientDescent can be run incrementally, for each new observations $(\mathbf{x}, y)$ acquired by the learner, instead of globally on the entire collection of data $\mathcal{D}$ (as in the pseudo-code of Algorithm 10.2). The former is called the *stochastic gradient descent* (SGD) mode, while the latter is the *batch* mode. SGD is computationally demanding. Moreover, it may drive the estimate in a noisy way, leading to a high error variance. But it is favorable for avoiding local minima, to which the batch mode is prone. Alternatively, in a *mini-batch* mode, the algorithm is repeatedly run on randomly sampled subsets of $\mathcal{D}$, which offers a good compromise.

Let us summarize and underline the main points seen so far in this section:

- It is easy to learn from a data collection of observed matching pairs $\mathcal{D} = \{(\mathbf{x}, y)\}$ by approximating the relationship in $\mathcal{D}$ with a parametric function $y = f_\theta(\mathbf{x})$.
- The optimal parameters $\boldsymbol{\theta}^* = \text{argmin}_\theta\{Loss(f_\theta)\}$ can be computed with the GradientDescent algorithm as long as f_θ is differentiable, with a caveat about local minima when $Loss(f_\theta)$ is not convex.
- The merits of the approach are
  - to generalize what has been observed in $\mathcal{D}$ to new predictions;
  - to easily compute a prediction of a y from a new $\mathbf{x}$;
  - to improve the entire function f_θ on each new observed pair, instead of just adding a new instance to the collection experienced by the learner.

Note however that for a pair $(\mathbf{x}^{(i)}, y^{(i)}) \in \mathcal{D}$, in general $f_\theta(\mathbf{x}^{(i)}) \neq y^{(i)}$: we generalizes and approximates the relation in $\mathcal{D}$, but f_θ is not a faithful table-lookup.

To sum up, we can model a training set $\mathcal{D}$ with a parametric function f_θ. Finding $\boldsymbol{\theta}^*$ given $\mathcal{D}$ is a *learning problem*. Symmetrically, finding $y = f_\theta(\mathbf{x})$ given $\mathbf{x}$ and a model f_θ is a *prediction* or *inference problem*.[8]

In RL we view $\mathcal{D}$ as being defined incrementally through acting and observing. When $\mathcal{D}$ is a set of training data given *a priori*, this is called *supervised learning*.

10.3.3 Parametric Q-learning

We just saw how to generalize a finite set of observations $\mathcal{D}$ by representing the corresponding relationship as a parametric function f_θ, and how to compute the appropriate values of the parameters. Let us see how this can be used in Q-learning.

Earlier (Section 10.2) we addressed RL using a large table to store $Q(s, a)$ for every s and a. This has two drawbacks: *(i)* it takes a huge amount of space, and *(ii)* it doesn't generalize, i.e., it doesn't provide a way to estimate Q for pairs (s, a) not seen before. This is like keeping just $\mathcal{D}$ instead of f_θ. Here, we want to develop a parametric function Q_θ that approximates Q, and adjust the value of $\boldsymbol{\theta}$ to provide the best fit to the observed rewards.

Parametric techniques are widely used for learning in classification and clustering problems. We can transpose them to action learning by seeking a parametric approximation of the relationship from the state space S, to the action space A. Such a transposition fits naturally to a learning scenario where the learner observes (state, action) pairs, as in learning from the demonstrations of a teacher (see Section 10.7). However, in reinforcement learning, the learner does not observe target values as the teacher's actions, but the rewards from its own actions.

In general, parametric RL seeks a parametric approximation of V, π, or Q, integrating the rewards r. In Q-learning, we parametrize the action-value as a function $Q_\theta(s, a)$, with a vector of parameters $\boldsymbol{\theta}$. Q_θ is assumed to be differentiable with respect $\boldsymbol{\theta}$. The optimal value $\boldsymbol{\theta}^* = \text{argmin}_\theta\{Loss(Q_\theta)\}$ is still computed with gradient descent. However the empirical loss is no longer defined on the difference between predicted and observed, since no observation is available here. The loss is defined with respect to the difference between Q_θ and an optimal Q^*_θ as given by the update rule Equation 10.4 for the Bellman Equation 10.3. Namely, for a single observation the target is:

$$y = r(s, a, s') + \max_{a'}\{Q_\theta(s', a')\}, \tag{10.7}$$

and the loss function is:

$$Loss(Q_\theta) = [y - Q_\theta(s, a)]^2. \tag{10.8}$$

Updating Q means updating its parameters $\boldsymbol{\theta}$. The gradient of $Loss(Q_\theta)$ is:

$$\nabla_\theta Loss(Q_\theta) = -(y - Q_\theta(s, a))[\ldots, \frac{\partial Q_\theta(s, a)}{\partial \theta_j}, \ldots]^\top$$

[8] Note that learning is an *inverse* problem, while prediction is a *direct* problem.

$$= -(y - Q_\theta(s, a))\nabla_\theta Q_\theta(s, a)$$

From y in Equation 10.7, the update rule of Section 10.3.2 for an element of θ is :

$$\theta_j \leftarrow \theta_j + \alpha[r(s, a, s') + \max_{a'}\{Q_\theta(s', a')\} - Q_\theta(s, a)]\frac{\partial Q_\theta(s, a)}{\partial \theta_j} \tag{10.9}$$

This can be expressed in a vector form, easily parallelizable, as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[r(s, a, s') + \max_{a'}\{Q_\theta(s', a')\} - Q_\theta(s, a)]\nabla_\theta Q_\theta(s, a) \tag{10.10}$$

Parametric Q-learning tries performing the task a given number of episodes. In each episode, it starts from an initial state randomly drawn from S_0, and runs over a *finite* number of steps. In each step, a selected action a is performed, the next state s' and a reward $r(s, a, s') \in \mathbb{R}$ are observed. The target y for this observation is computed, and the parameters are updated. The algorithm follows a policy given by Select; it is off-policy.

```
Parametric Q-learning
    initialize $\boldsymbol{\theta}$
    for each episode do
    |   randomly draw a starting state $s$ from $S_0$
    |   until episode termination do
1   |   |   $a \leftarrow$ Select$(s)$                    // selects $a \in Applicable(s)$
2   |   |   perform action $a$
3   |   |   observe resulting state $s'$ and reward $r(s, a, s')$
4   |   |   $y \leftarrow r(s, a, s') + \max_{a'}\{Q_\theta(s', a')\}$
5   |   |   $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha[y - Q_\theta(s, a)]\nabla_\theta Q_\theta(s, a)$
6   |   |   $s \leftarrow s'$
```

Algorithm 10.3. Parametric Q-learning algorithm.

The pseudo-code in Algorithm 10.3 updates the parameters at each observation, as discussed for the stochastic mode of GradientDescent. Alternatively, in a mini-batch mode we would perform updates with respect to the average difference to targets over randomly sampled subset of previous observations.

In summary, Parametric Q-learning involves four stages :

(i) *Parametrize* the action-value as a function Q_θ,
(ii) *Experiment* to acquire a set of pairs (s, a) and corresponding rewards r,
(iii) *Optimize* the parameters with gradient descent towards a vector of values $\boldsymbol{\theta}^* = \text{argmin}_\theta\{Loss(Q_\theta)\}$,
(iv) *Predict* good actions with $\max_a Q_\theta(s, a)$.

Stage *(i)* requires finding numeric features of the state and action spaces. Typically, characteristic functions of the states and actions are assembled into a vector of numeric

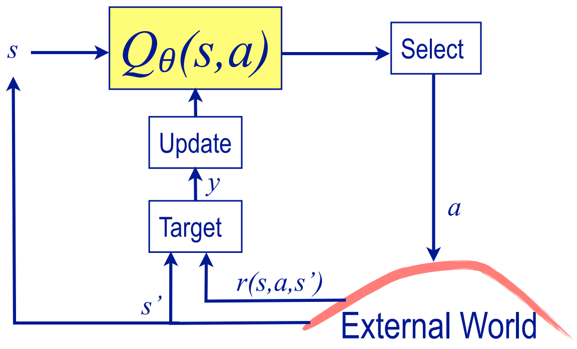


Figure 10.2. A schematic view of Parametric Q-learning

features $\boldsymbol{\phi} = [\phi_1, \ldots, \phi_n]^\top$, which allow the action-value function to be expressed parametrically, e.g., linearly as:

$$Q(s,a) = \sum_{1 \le i \le n} \theta_i \phi_i(s,a) = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(s,a)$$

Stages *(ii), (iii)* and *(iv)* are interleaved within incremental learning and acting.

Parametric Q-learning differs from memory-based Q-learning on the following points:

- An update following an experienced pair (s,a) does not change locally $Q(s,a)$ for that pair, it improves globally the function Q_θ.
- In an exploitation stage, Select(s) can easily predict a good action from $\max_a Q_\theta(s,a)$ without prior experience with the pair (s,a); i.e., the learner generalizes from what it experienced.
- In an exploration stage, the learner can steer an active learning strategy with more powerful heuristics than the ϵ-greedy rule of Equation 10.2. When looking an alternative to $\text{argmax}_a Q_\theta(s,a)$ for exploration, Select may seek an informative action that is in poorly sampled areas of Q_θ.

Seeking a good parametric function $Q_\theta(s,a)$ from states and actions to $\mathbb{R}$ is not straightforward. This is certainly easier when S and possibly A are metric spaces, e.g., vectors in $\mathbb{R}^n$, each component of which being a continuous state variable. This is illustrated in the following example with state and control variables in robotics.

When the state variables are essentially symbolic, meaningful numeric features of the states have to be sought and used in the parametrization. This leads to representing states in a metric space, as illustrated next.

Parametric RL requires first to parametrize the action-value function. For that neural networks happen to be excellent parametric function approximators, with powerful tools for learning the needed parameters. Let us first introduce briefly neural nets before considering their use value-based RL.

10.4 Neural Parametric Function Approximators

We mentioned earlier a few families of parametric functions f_θ that can be used to model a training set $\mathcal{D}$. Neural networks are also such a family of parametric functions. In RL, they are referred to a *neural function approximators*. They offer powerful optimization algorithms to find $\boldsymbol{\theta}^* = \text{argmin}_\theta\{Loss(Q_\theta)\}$), and allow for simple prediction of $f_\theta(x)$. Multilayered neural networks (also called Deep Neural Nets, DNN) can take as input high dimensional raw data for S. This simplifies the stage *(i)* (p. 237) for finding appropriate features.

Let us introduce briefly neural nets by focusing on simple feedforward networks for our needs in Deep reinforcement learning.

10.4.1 Simple Feedforward Neural Nets

Cells of neural nets. A neural network is an organized collection of computational units, called cells (or artificial neurons). A cell in a NN is an extension of linear regression. Recall that multivariable linear regression gives as output a scalar $\tilde{y} = \boldsymbol{\theta} \cdot \mathbf{x}$ (in vector notation). A neural cell is also a parametric function $f_\theta(x) = g(\theta \cdot x)$, where g is a nonlinear *activation* function.[9] Possible activation functions are, for example (see Figure 10.3):

- the *logistic* (or sigmoid) function: $g_1(z) = \frac{1}{1+e^{-z}}$,
- the *rectified linear* function (ReLU): $g_2(z) = \max\{0, z\}$,
- the *softplus* function: $g_3(z) = \log(1 + e^z)$.

All these functions are nondecreasing, i.e., their derivative are nonnegative. They have a threshold effect: they reduce their output for a negative or small input $\boldsymbol{\theta} \cdot \mathbf{x}$, and are neutral for large input.

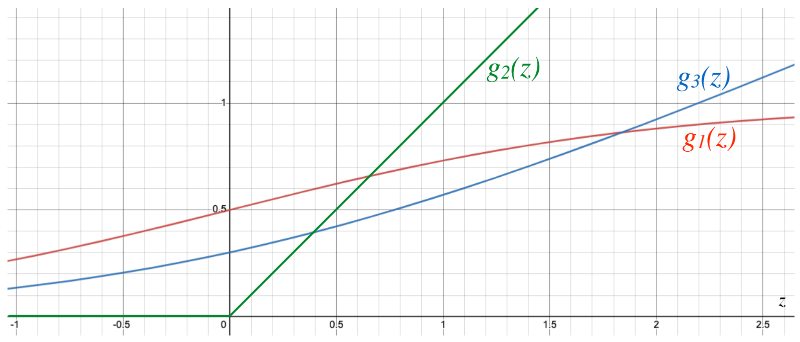


Figure 10.3. Examples of nonlinear activation functions.

A cell in a NN (pictured in Figure 10.4(a)) is a mapping from a vector to a scalar. It has as many parameters as its input vector, plus one for the bias, implicitly integrated into a dummy input in vector notation (as in Equation 10.6).

[9] By reference to a biological neuron, which activates when its total input stimulus is beyond a threshold.

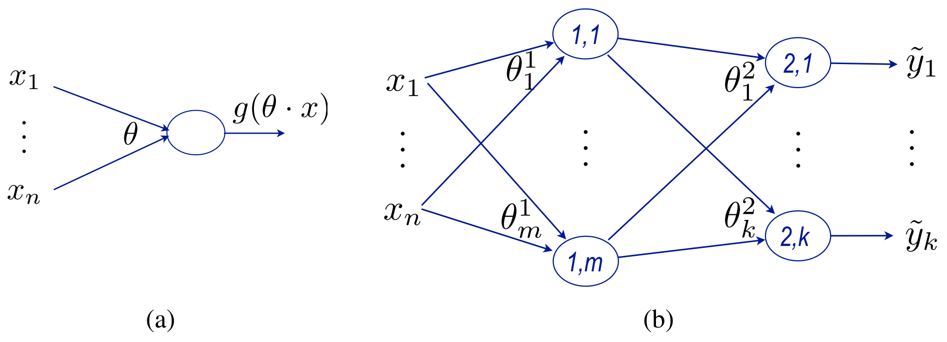


Figure 10.4. (a) a cell of a neural net; (b) a simple feedforward two layers NN. Bias terms are implicit.

Multilayered neural nets. A feedforward NN organizes several cells into layers. All cells at layer 1 take as common input the elements of a vector $\mathbf{x}$. The output of the cells at layer l are given as input to the cells at $l+1$. The last layer gives a global output vector $\tilde{\mathbf{y}}$ matching the input x. Intermediate layers, except for the first and last ones, are called *hidden* layers.

Consider the two layers NN in Figure 10.4(b), which has m cells in the first layer, and k in the second. Their parameters are the vectors $\boldsymbol{\theta}^1_1$ to $\boldsymbol{\theta}^1_m$, and $\boldsymbol{\theta}^2_1$ to $\boldsymbol{\theta}^2_k$. The input common to the cells in the first layer is the vector $\mathbf{x} = [x_1, \ldots, x_n]^\top$. Their respective output are m scalars which we can write collectively as a vector $[g(\boldsymbol{\theta}^1_1 \cdot \mathbf{x}), \ldots, g(\boldsymbol{\theta}^1_m \cdot \mathbf{x})]^\top$.

It is convenient not only to write the input and output of a layer of cells as vectors, but also to write their parameters as matrices. Let Θ^1 be an $(m, n+1)$ matrix whose *rows* are the vectors $\boldsymbol{\theta}^1_1$ to $\boldsymbol{\theta}^1_m$, that is, $\Theta^1_{(m,n+1)} = [\theta^1_{i,j}]$ is the matrix of the parameters of the first layer cells, $\theta^1_{i,j}$ being the j^{th} parameter of $\boldsymbol{\theta}^1_i$ connecting x_j to the j^{th} cell of the first layer.

In matrix notation (see Section B.2), the product of matrix $\Theta^1_{(m,n+1)}$ and vector $\mathbf{x}_{(n+1,1)}$ is a vector of dimension m: $g(\Theta^1 \times \mathbf{x}) = [g(\boldsymbol{\theta}^1_1 \cdot \mathbf{x}), \ldots, g(\boldsymbol{\theta}^1_m \cdot \mathbf{x})]^\top$. It is the output of the first layer. It has as many elements as cells in the first layer. We implicitly add to the output of the first layer $g(\Theta^1 \times \mathbf{x})$ a dummy element for the bias (as for the input x).

The same formulation is applied to the second layer. An output of this layer is: $\tilde{y}_i = g(\boldsymbol{\theta}^2_i \cdot g(\Theta^1 \times \mathbf{x}))$, for $1 \leq i \leq k$. Writing the parameters of the second layer as a matrix $\Theta^2_{(k,m+1)}$ whose rows are the vector $\boldsymbol{\theta}^2_1$ to $\boldsymbol{\theta}^2_k$, we get the global output of this network for the input x as the vector:

$$\tilde{\mathbf{y}} = f_\Theta(x) = g(\Theta^2 \times g(\Theta^1 \times \mathbf{x})) \tag{10.11}$$

where Θ is a short hand notation for the pair of matrices (Θ^1, Θ^2) with all the parameters.

This network has $m+k$ cells, which all use the same activation function g. It has $[(n+1)m + (m+1)k]$ parameters. The optimal value of these parameters

$\Theta^* = \text{argmin}_\Theta\{Loss(f_\Theta)\}$ can be computed as in the previous section. We can take for example the empirical loss as the mean squared error between the estimated $\tilde{\mathbf{y}}$ and the targeted $\mathbf{y}$ for a given pair $(\mathbf{x}, \mathbf{y})$, that is : $||f_\Theta(\mathbf{x}) - \mathbf{y}||^2$.[10] This gives, for the entire collection of matching pairs $\mathcal{D}$, the following $Loss$:

$$Loss(f_\Theta) = \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} ||f_\Theta(\mathbf{x}) - \mathbf{y}||^2 \tag{10.12}$$

Here too, the minimum of $Loss(f_\Theta)$ is computed from the equation $\nabla Loss(f_\theta) = [\frac{\partial Loss(f_\theta)}{\partial \theta^l_{i,j}}] = 0$, over all the parameters in the network. Gradient descent is the mean for estimating this minimum when g, hence f, are differentiable. In a multiple layer nets, it has to take into account the function compositions, as explained next.

10.4.2 Backpropagation Gradient Descent

The idea here is to optimize the network parameters for a training database. Given a target $\mathbf{y}$ and a network computed estimate $\tilde{\mathbf{y}} = f_\Theta(\mathbf{x})$, the problem is to change Θ such as to reduce the $Loss(f_\Theta)$. This is done as in Section 10.3.2 by making an update of Θ following the gradient vector. This update is computed first with respect to the last layer of the network, then it is back propagated step by step to the first layer.

Consider a feedforward network of L layers whose cells have the same activation function g. Let Θ^l be the matrix of parameters of the cells at layer l. Equation 10.11 generalizes to:

$$\tilde{\mathbf{y}} = f_\Theta(\mathbf{x}) = g(\Theta^L \times g(\Theta^{L-1} \times g(\ldots g(\Theta^1 \times \mathbf{x})) \tag{10.13}$$

The partial derivatives of $Loss(f_\theta)$ are computed using the chain rule for the derivative of a composite function. Namely, for a composed function $g(z(\theta))$, the derivative is $\frac{\partial g}{\partial \theta} = \frac{\partial g}{\partial z}\frac{\partial z}{\partial \theta}$. Now, this network of L layers and n cells per layer has $n^2 L$ parameters. Working the derivative chain rule on Equation 10.13 through all the layers and parameters of the network can be quite complicated. It needs to be efficiently organized. The Backpropagation gradient descent algorithm does it over two paths:

- a forward path which compute $\tilde{\mathbf{y}}$ from layer 1 to L with Equation 10.13 for a given x, and caches the intermediate results needed for the second path, and
- a backward path which propagate the derivatives for the loss of $||\tilde{\mathbf{y}} - \mathbf{y}||^2$ and the updates of all the parameters from layer L to 1.

To simplify the computation, we'll introduce a set of vectors: $\mathbf{x}^l$, for $1 \le l \le L$. The $\mathbf{x}^l$ will allow us to decompose Equation 10.13 into L operations:

- $\mathbf{x}^1 = \mathbf{x}$,
- $\mathbf{x}^{l+1} = g(\Theta^l \times \mathbf{x}^l)$ is the output vector of layer l.

The final output of the network is the vector $\mathbf{x}^{L+1} = f_\Theta(\mathbf{x}) = \tilde{\mathbf{y}}$.

In order to compute the partial derivative of $||\tilde{\mathbf{y}} - \mathbf{y}||^2$ with respect to the parameters let us focus on the last layer for a single parameter:

[10]Other loss functions are, e.g., the cross entropy loss: $Loss = \sum_i \tilde{y}_i log y_i$; see Exercise 10.2.

$$\|\tilde{\mathbf{y}} - \mathbf{y}\|^2 = \sum_i [g(\boldsymbol{\theta}_i^L \cdot \mathbf{x}^L) - y_i]^2 = \sum_i [g(\sum_j \theta_{i,j}^L x_j^L) - y_i]^2$$

Hence, considering the loss for a single observation $(\mathbf{x}, \mathbf{y})$:

$$\frac{\partial \|\tilde{\mathbf{y}} - \mathbf{y}\|^2}{\partial \theta_{i,j}^L} = 2 \sum_i [g(\boldsymbol{\theta}_i^L \cdot \mathbf{x}^L) - y_i] \frac{\partial [g(\sum_j \theta_{i,j}^L x_j^L) - y_i]}{\partial \theta_{i,j}^L}$$

The derivative of the composed $g(z(\boldsymbol{\theta}))$ give the last term in the above equation as $\frac{\partial g(\sum_j \theta_{i,j}^L x_j^L)}{\partial z} x_j^L = g'(\sum_j \theta_{i,j}^L x_j^L) x_j^L$. Hence, the update of $\theta_{i,j}^L$ is:

$$\theta_{i,j}^L \leftarrow \theta_{i,j}^L - \alpha \sum_i [g(\boldsymbol{\theta}_i^L \cdot \mathbf{x}^L) - y_i] g'(\sum_j \theta_{i,j}^L x_j^L) x_j^L \tag{10.14}$$

In matrix notations we can extend this computation over all the parameters Θ^L of layer L in a more readable (and parallizable) way. This is done by defining a vector $\boldsymbol{\delta}^{L+1}$ and an update rule for Θ^L which extends the rule for a single parameter $\theta_{i,j}^L$:

$$\begin{aligned} \boldsymbol{\delta}^{L+1} &= (\mathbf{x}^{L+1} - \mathbf{y}) \times g'(\Theta^L \times \mathbf{x}^L) \\ \Theta^L &\leftarrow \Theta^L - \alpha[\boldsymbol{\delta}^{L+1} \otimes (\mathbf{x}^L)^\top] \end{aligned} \tag{10.15}$$

Note that the outer product gives a matrix of the same dimension as Θ^L.

Performing this computation one step back on a parameter $\theta_{i,j}^{L-1}$ of layer $L-1$ shows how to define $\boldsymbol{\delta}^L$ as a function of $\boldsymbol{\delta}^{L+1}$ (see Exercise 10.1) and the update rule for Θ^{L-1}:

$$\begin{aligned} \boldsymbol{\delta}^L &= [(\Theta^L)^\top \times \boldsymbol{\delta}^{L+1}] \times g'(\Theta^{L-1} \times \mathbf{x}^{L-1}) \\ \Theta^{L-1} &\leftarrow \Theta^{L-1} - \alpha[\boldsymbol{\delta}^L \otimes (\mathbf{x}^{L-1})^\top] \end{aligned}$$

This extends backward over the entire network until δ^2 and Θ^1 to update the entire matrix Θ.

Backpropagation is a sequence of a forward path to compute the $\mathbf{x}^l$ vectors, then a backward path to compute the $\boldsymbol{\delta}^l$ vectors and to update on the Θ^l matrices (Algorithm 10.4).

Backpropagation can be run, as the gradient descent, in different modes:

- in a stochastic mode, incrementally on each new pair $(\mathbf{x}, \mathbf{y})$,
- in a batch mode, over all data pairs $(\mathbf{x}, \mathbf{y})$ in $\mathcal{D}$, or
- in a mini-batch mode, over a subsets of pairs randomly drawn from D. In this mode the term δ^{L+1} in Equation 10.15 is summed and averaged out over for all pairs in a mini-batch, then propagated just once (to be illustrated in Deep Q-learning).

```
Backpropagation($(\mathbf{x}, \mathbf{y})$)
    $\mathbf{x}^1 \leftarrow \mathbf{x}$
    for $l = 1$ to $L$ do                                          // Forward path
        $\mathbf{x}^{l+1} \leftarrow g(\Theta^l \times \mathbf{x}^l)$
    $\delta^{L+1} \leftarrow (\mathbf{x}^{L+1} - \mathbf{y}) \times g'(\Theta^L \times \mathbf{x}^L)$
    $\Theta^L \leftarrow \Theta^L - \alpha[\delta^{L+1} \otimes (\mathbf{x}^L)^\top]$
    for $l = L$ to $2$ do                                          // Backward path
        $\delta^l \leftarrow [(\Theta^l)^\top \times \delta^{l+1}] \times g'(\Theta^{l-1} \times \mathbf{x}^{l-1})$
        $\Theta^{l-1} \leftarrow \Theta^{l-1} - \alpha[\delta^l \otimes (\mathbf{x}^{l-1})^\top]$
```

Algorithm 10.4. Back propagation algorithm for a feedforward neural net with incremental update from a single observation $(\mathbf{x}, \mathbf{y})$.

In summary, a neural function approximator is a powerful mean for estimating a complex input/output relation from a collection of pairs (input, target output) examples using Backpropagation. In many ways, the use of DNN in RL is similar to their use in supervised learning, where $\mathcal{D}$ is an apriori given collection of training data (the most frequent application case of DNN). The specifics of Deep RL are discussed next.

10.5 Deep Value-Based RL

We introduced earlier parametric Q-learning to handle large state spaces, and neural nets to efficiently find the parameters of a model. Let us combine here the two in an algorithm called Deep Q-learning.

Deep Reinforcement Learning is a parametric Q-learning that uses multilayered neural networks. It's a model-free RL: the learner has no prior model of γ and Pr. It observes states and rewards from its actions. Its learning method relies on the four stages summarized page 237.

In principle, what we saw for Parametric Q-learning in Section 10.3.3 applies to the case where Q_θ is estimated with a neural net. Recall that the target value for Q-learning is $y = r(s, a, s') + \max_{a'}\{Q_\theta(s', a')\}$ (Equation 10.7). Hence the difference between the target y and the estimated $\tilde{y} = Q_\theta(s, a)$ is given as in the update rule of Equation 10.9. A neural net Q-learning algorithm uses the Backpropagation procedure to update the network parameters.

10.5.1 Network Organization for Deep RL

To work out these principles into a pseudo-code for Deep Q-learning, let us first discuss a few important network organization issues.

Network Input and Output. A first issue is about the input/output of the neural net. A direct transposition of Section 10.3.3 would take s and a (or more precisely an adequate coding of s and a) as the network input and require $Q_\theta(s, a)$ as a scalar output, that is to make the function f_θ computed by the network be $f_\theta = Q_\theta$. An alternative is to input s and to output a vector of values $[Q_\theta(s, a_1), \dots, Q_\theta(s, a_m)]^\top$ for all ground actions a_i of the domain. The latter approach, used in Deep Q-learning, is computationally more efficient.

Target Values. Another important point is that the target value y computed with Equation 10.7 uses the neural net. Hence it depends on the network parameters. While this is not a problem for Q-learning, in neural nets, the target y should be taken as independent of $\boldsymbol{\theta}$ (see Equation 10.12).[11] This can be dealt with by updating the parameters at an iteration k of Algorithm 10.3 with a target y that uses the parameters of the previous iteration $k-1$. But in that case, an update that increases Q at iteration k may increases y at the next iteration, leading possibly to oscillations. The intuition, confirmed empirically, is that this approach introduces a close link between the estimates $\tilde{y}$ and the targets y, resulting in an instable or even a non-converging learning process. The solution for diminishing this link is to use two networks.

The idea is to copy the original NN after a number υ of updates, and use it to compute the target y for the current s, in order to update only the original network with respect to this target. This copy is called the *target network*. Targets will use stable and older version of $\boldsymbol{\theta}$ and will not be affected by ongoing updates. This "delay" υ updates does not need to be too large, just enough to avoid oscillations.[12]

Experience replay and mini-batch updates. A related issue is about the stochastic mode of back propagation. Recall that the online Q-learning formulation of Section 10.3.3 performs learning updates incrementally after each new experience, whose result is then discarded. But one may store a series of experiences then re-use their results in batch updates, as seen in back propagation which can be performed in batch or in stochastic mini-batches. Applications in games and a few other domains show that it may be better to cache a series of the last N steps, possibly over several episodes, in a buffer called a *replay-memory*, and perform updates on random mini-batches drawn from this replay memory. This is motivated by the following points:

- successive steps in an episode are generally correlated, in particular when Select returns $\mathrm{argmax}_a\, Q$ (exploitation is more frequent then exploration), hence the assumption of independent and identically distributed observations does not hold,
- updates can be less frequent than observations, and in mini-batches, a given observation may be potentially used for several updates.

The random selection of a mini-batch in the replay memory does not need to be uniform; it may follow a more elaborate strategy, such a *prioritized sweeping*. In the latter, a queue is maintained of every state-action pair whose estimated value changes

[11] In supervised learning, the target is given *a priori* before learning starts.

[12] This is akin from a control theory methods for handling instabilities: introduce a *delay* such as to increase the time constant of a system. The equivalent of a delay here is lag between the updates of the targets and the estimates.

if updated, prioritized by the size of the change. When the top pair in the queue is updated, the effect on each of its predecessor pairs is computed. If the effect is greater than some threshold, then the pair is inserted in the queue with the new priority.

10.5.2 Deep Q-learning Algorithm

The Deep Q-learning is a parametric Q-learning algorithm using neural nets that follows the above ideas. It seeks to synthesize a near-optimal function $Q_\theta : S \rightarrow A$, and hence a policy for the task at hand $\pi(s) = \mathrm{argmax}_a Q_\theta(s,a)$. The learning proceeds by repeating (with a simulator or in the real world) a number of trials for that task. In each trial, called an *episode*, the learner starts from some initial state, randomly drawn from a given set S_0 of possible initial states. As said earlier (Section 10.2.2), we assume that each trial terminates after a finite but variable number of steps, when the task finishes. In each step, an action a is performed, the next state s' and a reward $r(s,a,s')$ are observed. The algorithm is off-policy: the selected action (line 1) is not necessarily the current policy given by $\mathrm{argmax}_a Q_\theta$. Recall that this allows Select to combine exploitation with exploration.

```
Deep Q-learning
    initialize $\boldsymbol{\theta}$ for network $[Q_\theta]$ and replay-memory $\mathcal{R}_M$
    $[\hat{Q}_{\theta^-}] \leftarrow [Q_\theta]$                          // target network
    for all episodes do
        randomly draw a starting state $s$ from $S_0$
        until episode termination do
1           $a \leftarrow$ Select$(s)$                           // selects $a \in Applicable(s)$
            perform action $a$
            observe resulting state $s'$ and reward $r(s,a,s')$
            push$((s,a,s',r(s,a,s')),\mathcal{R}_M)$             // FIFO replay memory
            $\mathcal{B} \leftarrow$ set of $k$ tuples uniformly sampled from $\mathcal{R}_M$
            $\boldsymbol{\delta} \leftarrow [0,\ldots,0]$
            forall tuples $(s,a,s',r(s,a,s')) \in \mathcal{B}$ do
2               $y \leftarrow r(s,a,s') + \max_{a'}\{\hat{Q}_{\theta^-}(s',a')\}$
3               $\boldsymbol{\delta} \leftarrow \boldsymbol{\delta} + 1/k[y - Q_\theta(s,a)]\nabla_\theta Q_\theta(s,a)$
4           $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\boldsymbol{\delta}$            // update with Backpropagation
            $s \leftarrow s'$
5           every $v$ steps reset $[\hat{Q}_{\theta^-}] \leftarrow [Q_\theta]$        // update target net
```

Algorithm 10.5. Deep Q-learning algorithm.

Deep Q-learning uses a main neural net, denoted $[Q_\theta]$. Its input is a coding of the current state s. Its output is a vector $[Q_\theta(s,a_1),\ldots,Q_\theta(s,a_m)]^\top$ over available actions. This network is used to compute the Q_θ vector and to learn its parameters. The network $[Q_\theta]$ is periodically copied into a *target network* denoted $[\hat{Q}_{\theta^-}]$, which is used solely to compute the target value $y = r(s,a,s') + \max_{a'}\{Q_{\theta^-}(s',a')\}$ with the parameters θ^-, corresponding to previous values of the parameters of $[Q_\theta]$. This allows avoiding oscillations due to a target too close to the estimate (see p.244).

The algorithm keeps a replay-memory $\mathcal{R}_M$ as a FIFO list recording the last N steps, where new observations remplace old ones. Hence $\mathcal{R}_M$ covers successive episodes, possibly several if N is taken large enough. A mini-batch $\mathcal{B}$ of k tuples is sampled uniformly from the replay-memory $\mathcal{R}_M$. More elaborate sampling strategy are feasible, e.g., *prioritized sweeping*.

The parameters of $[Q_\theta]$ are updated (line 4) with Backpropagation algorithm, as in Equation 10.10, with an error term averaged out over the k tuples in the mini-batch $\mathcal{B}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha}{k} \sum_{(s,a,s',r)\in\mathcal{B}} [y - Q_\theta(s,a)]\nabla_\theta Q_\theta(s,a)$$

The update term $\boldsymbol{\delta}$ is computed in the **forall** loop. Note that y and $Q_\theta(s,a)$ are scalar (from respectively the output of $[\hat{Q}_{\theta^-}]$ and $[Q_\theta]$ for action a), while $\nabla_\theta Q_\theta(s,a)$ and $\boldsymbol{\delta}$ are vectors of the same dimension as $\boldsymbol{\theta}$. Parameter updates in the pseudo-code are performed at each observation, but they can be less frequent since updates are computationally costly.

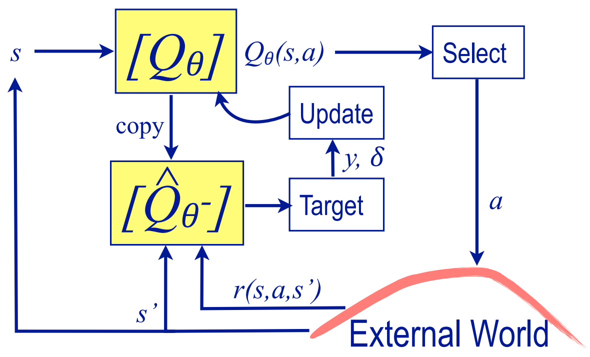


Figure 10.5. A schematic view of Deep Q-learning showing the main neural net $[Q_\theta]$ and the target network $[\hat{Q}_{\theta^-}]$. The replay-memory and mini-batch buffers are not depicted.

Algorithm Deep Q-learning, as all the other Q-learning algorithms seen so far in this chapter, are intended for finite state and action spaces. However, Deep Q-learning can easily handle a high dimensional state space, such as images in video-game applications.[13] Images are handled with Convolution Neural Nets. Deep Q-learning can also handle a continuous state space such as vectors in $\mathbb{R}^n$. However, the action space is necessarily finite and rather low-dimensional, since its dimension impacts the network size. We'll see how to cope with a high dimensional and a continuous action space in sections 10.6.2 and 22.1 .

[13]An image of $n \times m$ pixels, each ranging over k values, covers a space in $k^{n\times m}$.

10.5.3 Network Engineering for Deep RL

Deep Q-learning integrates the ingredients seen so far for value-based reinforcement learning together with the power of neural nets for learning and incrementally updating a parametric function Q_θ. It has been used with very good performances in video games and similar applications. However, its successful deployment requires adequate choices and careful tuning of several parameters. Let us discuss the main engineering issues related to this algorithm.

Controlling the learning rates. The setting of the learning rate α (in Backpropagation) is delicate issue. Imagine moving back and forth along a U-shaped curve (when *Loss* is convex) towards its minimal value. Large steps may trigger oscillations around the minima, while small steps will slow down learning. The strategy of reducing α as learning progresses has to be informed about the shape of the *Loss* function to be minimized. Furthermore, learning does not progress for all the parameters at the same rate. One may consider a different learning rate for each parameter, but that would be too complicated. A more easily informed tuning strategy sets a different value of α for each layer of the DQN. A few methods, such as *RMSProp*, combine this tuning with mini-batch selection methods.

Architecting the network. The network architecture is a critical issue. Significant research in NN learning is devoted to architecting networks for specific needs. A coverage of main NN architectures is beyond the scope of this chapter. The reader should however keep in mind that there are many network types, other than the feedforward nets introduced earlier, that can be relevant to a learning actor, among which Convolution Nets are very popular. There are also many different loss functions, adapted to specific needs, e.g., the cross entropy loss for classification problems (see Exercise 10.2).

Of particular interest are convolution Neural Nets (CNN), which are prescribed when images are taken as input or as part of the network input. For example, video game applications take generally as input the raw color bitmap screen of the game; robotics or self-driving car applications may take the output of cameras and lidars as part of the input state s. In general, a convolution is an operation, denoted $f * g$, on two functions which modifies the shape of f with respect to that of g. In image processing, a convolution filter is an operation applied to sequences of overlapping *windows* in an image to extract salient features that are robust to translation and scaling transformations. A CNN organizes the connectivity of the cells of its first layers as such windows on an input image. The remaining layers can be fully connected and used for classification or estimate of the Q values. Other NN architectures are briefly discussed Section 10.9.

Hyperparameters. These and other issues lead to several choices and settings, usually referred to as the *hyperparameters* of a network, which can affect the performances of a DQN implementation. Among these are:

- ϵ in the ϵ-greedy Select function,

- the delay ν of the target network,
- the sizes of the *Replay-memory* and the *Mini-batch*,
- the parameters for controlling the learning rates α,
- the number of episodes.

The use of a simulator entails other settings such as initializations, random restarts, possibly a bound on the length of tried trajectories, and experimental verifications.

10.6 Policy-Based RL

We move here from the value-based techniques, seen so far in this chapter, to policy-based techniques. The dual role of value and policy has been illustrated in several planning algorithms in Chapter 9, in particular in Value Iteration and Policy Iteration. A value function gives a policy and symmetrically a policy correspond to a value function. In RL, the main differences between value-based and policy-based approaches are the following:

- A value-based RL approach is a search in the *value space*, towards a good approximation of Q^*_θ. A policy-based RL is a search in the *policy space*; it seeks to progressively improve a policy π towards an optimal π^*, taking into account the observed rewards of the trial-and-error experiments.
- A parametric value-based algorithm as Deep Q-learning can handle a high-dimensional state space, it is limited to a finite low-dimensional action space since the number of ground actions is a dimension of the network. A policy-based RL is to able to handle continuous and high-dimensional action spaces.

There are generally stronger convergence guaranty results for policy-based approaches than for value-based one, although the latter in practice can be faster. But it is easier to integrate and benefit from prior knowledge in the policy-based techniques than in the value-based ones.

Let us discuss the principles of policy-based approaches before proposing a particular algorithm.

10.6.1 Principles of Policy-Based RL

A search in the policy space has already been illustrated for planning with the Policy Iteration algorithm. Recall that this search involves two successive stages:

- a policy evaluation stage and
- a policy improvement stage.

We also discussed in Section 9.1.2 the *Generalized Policy Iteration* scheme, which interleaves the stages of *partial* policy evaluation and improvement.

We rely here on Generalized Policy Iteration for policy-based RL. This scheme allows evaluating then improve a policy locally and incrementally, after each learning trial or mini-batch of a few trials. Policy-based RL also uses parametric approaches to approximate a policy and its value with parametric functions. These functions are conveniently learned and expressed as neural function approximators.

Let us consider how to perform the two evaluation and improvement stages:

- The policy evaluation stage cannot rely on the planning methods seen in Section 9.1.2 for evaluating V^{π}, since in model-free RL we do not know the domain model. Instead, we can estimate the value V^{π} or the action-value Q^{π} with local Bellman-updates, as seen in the value-based approaches.
- The policy improvement stage updates the policy with respect to the updated value or action-value functions. This might be done with a greedy maximisation over the set of actions. But with a high-dimensional continuous action space, this would require a complex maximisation over the action space at every update step. Rather than maximizing, a simpler method updates the policy parameters in the direction of the expected gradient of V^{π} or Q^{π}, towards a maximal policy.

These are the main principles of policy-based RL: a policy evaluation stage with local *Bellman-updates*, and a policy improvement stage with *gradient ascent*.[14] Working out these principles into a mathematically sound and efficient RL algorithm is however quite tricky. It involves several choices, e.g., between learning a deterministic policy or a stochastic policy (the latter maps a state to a parametric probability distribution over the action space), or between an on-policy or an off-policy strategy. Let us illustrate a deterministic policy gradient algorithm with an off-policy exploration. We'll discuss its advantages later.

10.6.2 A Deterministic Policy Gradient Algorithm

Consider an MDP task with high dimensional possibly continuous state and action spaces; typically $S \subseteq \mathbb{R}^n$ and $A \subseteq \mathbb{R}^m$. A policy-based RL for this task parametrize the policy as well as the action-value function with two sets of parameters. Let $\boldsymbol{\pi}_\theta : S \rightarrow A$ be a safe deterministic policy for that task parametrized with $\boldsymbol{\theta}$, and $Q_\omega(s,a)$ an approximation of the action-value function for the policy π_θ parametrized with $\boldsymbol{\omega}$. The Policy Gradient algorithm uses two neural nets referred to as the *critic* and the *actor* nets:

- a *critic* network, denoted $[Q_\omega]$, is used for the policy evaluation stage. It learns an approximation of the action-value function $Q_\omega(s,a)$, $\boldsymbol{\omega}$ being the critic's vector of parameters. It takes vectors s and a as input and gives a scalar as output.
- an *actor* network, denoted $[\boldsymbol{\pi}_\theta]$, is used for the policy improvement stage. It learns a parametric policy $\boldsymbol{\pi}_\theta$, $\boldsymbol{\theta}$ being the actor's parameters. It takes as input the state vector $s \in \mathbb{R}^n$, and gives as output the action vector $a = \boldsymbol{\pi}_\theta(s) \in \mathbb{R}^m$.

These two network learn in parallel, each is updated with data from the other. For a trial $(s, a, s', r(s, a, s'))$, the update-rules for these two nets are respectively:

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \alpha_\omega [r(s,a,s') + Q_\omega(s', \boldsymbol{\pi}_\theta(s')) - Q_\omega(s,a)] \nabla_\omega Q_\omega(s,a) \tag{10.16}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \nabla_\theta \boldsymbol{\pi}_\theta(s) \times \nabla_a Q_\omega(s, \pi_\theta(s)) \tag{10.17}$$

[14]This is a gradient ascent not descent: we are maximizing a value, not minimizing a loss.

The rule 10.16 is the policy evaluation update of the critic network. It is similar to what we saw earlier. The main difference is that the target $y = r(s,a,s') + Q_\omega(s', \boldsymbol{\pi}_\theta(s'))$ is defined with respect to the current policy $\boldsymbol{\pi}_\theta$ from the actor network, not with respect to $\text{argmax}_a\, Q_\omega$.[15] Here $\nabla_\omega Q_\omega$ is a vector of the same dimension as $\boldsymbol{\omega}$; $[y - Q_\omega(s,a)]$ is a scalar.

The rule 10.17 is the gradient ascent update of the actor net. It implements the product of a matrix $\nabla_\theta \boldsymbol{\pi}_\theta(s)$, of dimension (p,m), with a vector of dimension m, for p the number of parameters of the actor net and m the dimension of the action space. Note that $\nabla_a Q_\omega(s, \pi_\theta(s))$ is the gradient vector for the critic net with respect to its action vector input (not the gradient with respect to the network parameters $\nabla_\omega Q_\omega(s,a)$). Since $\boldsymbol{\pi}_\theta$ is a vector of dimension m, $\nabla_\theta \pi_\theta(s)$ is a Jacobian matrix where the i^{th} column is the gradient with respect to $\boldsymbol{\theta}$ of i^{th} action component of the policy π.

```
Policy Gradient
    initialize the actor and critic networks [$Q_\omega$] and [$\boldsymbol{\pi}_\theta$]
    for all episodes do
        randomly draw a starting state $s$ from $S_0$
        until Termination do
1           $a \leftarrow$ Select($s$)                                   // selects $a \in Applicable(s)$
            perform action $a$
            observe resulting state $s'$ and reward $r(s,a,s')$
            $y \leftarrow r(s,a,s') + Q_\omega(s', \boldsymbol{\pi}_\theta(s'))$
2           $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \alpha_\omega [y - Q_\omega(s,a)] \nabla_\omega Q_\omega(s,a)$        // update critic net
3           $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \nabla_\theta \pi_\theta(s) \times \nabla_a Q_\omega(s, \pi_\theta(s))$     // update actor net
            $s \leftarrow s'$
```

Algorithm 10.6. Policy Gradient deterministic Policy Gradient algorithm.

The Deterministic Policy Gradient implements these two rules (lines 2 and 3) incrementally after each trial with respect to the observed reward and resulting state. These two rules use different learning rates α_ω and α_θ Note that the exploration strategy is off-policy according to a Select function (line 1), which may blend the current policy $\boldsymbol{\pi}_\theta$ with random noise.

An important issue with this algorithm is that the evaluation stage does not strictly speaking evaluate the current policy: Q_ω is an approximation estimate of the action-value for π_θ. How to make sure that this estimate does not bias the gradient ascent updates of Equation 10.17? Fortunately, if some *compatibility conditions* are met, then policy gradient respond to this concern. These conditions involve a linear function approximator that estimates the action-value from features of the policy π_θ. They can be met with a modification of the update rules 10.16 and 10.17 (see paragraph 10.9). There are additional issues similar to those discussed in Deep Q-learning, to which we'll come back in Chapter 22 for learning sensory-motor skills.

[15]Recall that this is to avoid the maximization of the action space.

10.7 Aided Reinforcement Learning

RL requires guidance about desirable behaviors, which may not be available or rather sparse. Aided reinforcement learning supplements RL with means to help the learner progress. These means are for example a hierarchical break down of the task to learn into subtasks, or additional informations about the reward feedback received after an action trial. The former is the topic of Hierarchical Reinforcement Learning, briefly discussed on page 261 and further developed in Section 16.2 about learning hierarchical refinement methods. This section focuses on aids to RL through rewards, demonstrations and advices.

Previous RL algorithms indicates "observe reward $r(s, a, s')$". But rewards are seldom directly given by the environment and observable in every state. One must provide the means to estimate the progress of the task from what is perceived. Sometimes a function $r(s, a, s')$ derives naturally from the features of simple tasks. This is the case, for example, for domains that have "ideal" states and a deviation distance from these states, e.g., the deviation from equilibrium for a cartpole stabilization task, or the distance from the target for a tracking task. In a process maintenance problem, such as keeping tidy an area, one may decompose the reward into local returns over subareas and beneficial activities.

In complex tasks, defining a meaningful reward function can be difficult. For example, specifying the reward function for a robotics search-and-rescue task, or for a car driving task, is complicated. In goal reachability tasks, it is easy to check that a goal has been reached (as long as it is observable). But it is hard to appreciate in the middle of the task how much an action is progressing towards a goal, and rewarding it consequently in the learning process. Defining rewards as zero everywhere and 1 at goal states is theoretically acceptable, but seldom practical. These are called *sparse rewards* domains. They do not give much feedback to the leaner and make the convergence of RL very slow.

Sparse rewards can be alleviated by defining heuristics allowing to *shape* rewards for the domain and task at hand (Section 10.7.1). In *imitation* or *apprenticeship* learning, aids can be given by a teacher as demonstrations or advises, from which the apprentice may extract a guiding policy or appropriate rewards as a parametrized function r_θ, e.g., in a supervised learning framework (Section 10.7.2). Similarly, in *inverse* RL, the reward function is synthesized from a teacher's demonstrations (Section 10.7.3).

10.7.1 Reward Shaping and Learning Heuristics

As introduced in Section 8.3.2, shaping refers to a general property that allows modifying the cost or reward parameters of a domain without changing its optimal solution. Consider the transformation of an SSP domain Σ into a domain Σ' identical to Σ except for the cost function, replaced by: $\text{cost}'(s, a, s') = \text{cost}(s, a, s') - h(s) + h(s')$, for any function $h : S \to \mathbb{R}$. It is easy to check that the function $Q'(s, a) = Q(s, a) - h(s)$ meets the Bellman equation for Σ', i.e.,

$$Q'(s,a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s,a)[\text{cost}'(s,a,s') + \min_{a'}\{Q'(s',a')\}]$$

Hence, the optimal policy π'^* for Σ' is:

$$\pi'^*(s) = \operatorname{argmin}_a Q'(s,a) = \operatorname{argmin}_a [Q(s,a) - h(s)] = \operatorname{argmin}_a Q(s,a) = \pi^*(s).$$

Consequently, Σ and Σ' have the same optimal policy regardless the function h used to *shape* the cost. The same argument applies to the dual formulation which seeks to maximize the total expected rewards. It also applies to indefinite horizon MDP with a discounted shaping, i.e., $\text{cost}'(s,a,s') = \text{cost}(s,a,s) - h(s) + \delta h(s')$, with $0 \leq \delta \leq 1$.

Shaping can be very convenient in a goal reachability task. Recall that the immediate reward of an action can be meaningless while its influence on the task considerable. Consider a domain with sparse rewards: $r(s,a,s') = 0$ everywhere, except for goal states, for which the reward is 1. If the designer knows about a domain-independent or domain-specific heuristic function $h : S \to \mathbb{R}$ estimating how close a state s is to a goal, this heuristic can be used to shape the reward and guide the learner. If no such heuristic is known *a priori* but a domain simulator is available, then a problem-specific parametrized heuristic h_θ can be learned, using a planning algorithm from Chapter 9, then a NN estimator, trained on the $V(s)$ values obtained by such a planner. A heuristic learning approach can benefit from a sampling strategy of the domain, together with the Monte Carlo rollout techniques seen in Section 9.5.4.

Note that when h is a monotone heuristic (see Definition 9.8), shaping decreases the cost. In the dual formulation, it increases the reward. Note also that the value function V plays the role of a heuristic function in planning (see Section 9.2). Symmetrically, a heuristics estimates a value function, which can be used as initialization in RL.

Finally, in some cases a user may find it easier to provide its guiding knowledge informally in natural language, instead of a heuristic function h. Reward shaping in natural language is getting more interest with the development of Large Language Models (see Section 23.2.3).

10.7.2 Imitation Learning

In imitation learning, the learner has a reward function. In addition, it receives an aid from a teacher in the form of demonstrations about what to do for the task at hand in some states. These demonstrations are a collection of trajectories $\mathcal{D} = \{\sigma_1, \ldots, \sigma_N\}$, where a trajectory is a finite sequence of pairs $\sigma_j = \langle (s_{j_1}, \pi_d(s_{j_1})), \ldots, (s_{j_m}, \pi_d(s_{j_m})) \rangle$. In a simplified formulation of imitation learning, the learner starts with the demonstrated policy π_d, tries to generalize π_d into a policy over S, then improve this policy from the observed rewards of its actions.

We just saw how to generalize a demonstrated policy π_d with a parametric function π_θ using a supervised learning approach. A DNN with s as input and π as output is trained on all instances of π_d in $\mathcal{D}$ in order to estimate a function π_θ. But such an estimate might not be satisfactory. Indeed, a teacher's time is costly; one can only expect sparse demonstrations given the typical size of S and A. Unless the domain is very simple, the generalization from π_d to π_θ will be too brittle. It needs to be improved by the learner.

In a model-free RL approach, one may use π_θ in the Select procedure of Algorithm 10.3 to guide the exploration and learning of Q_θ. Select(s) will choose among three options:

$$\text{Select}(s) = \begin{cases} \pi_\theta(s) & \text{when confidence in } Q_\theta \text{ is low,} \\ \text{argmax}_a\{Q(a)\} & \text{when confidence in } Q_\theta \text{ is higher,} \\ \text{random alternative } \ a' & \text{with probability } \epsilon. \end{cases} \tag{10.18}$$

The initial set of trials will be focused on actions selected with π_θ. When more confidence on a parametrized Q_θ is reached, the learner would more frequently select $\text{argmax}_a Q(s,a)$, while exploring occasionally other informative trials.

In a model-based RL framework, the learner acts initially according to π_θ. It acquires statistics about the domain to estimate its model γ and Pr, from which it computes V^{π_θ} (with Equation 8.1), which estimates how good is the policy π_θ generalized from the teacher demonstrations. At that point, the learner can improve locally π_θ from its experience, using Proposition 9.1. Notice that the idea is not to stay as close as possible to π_d, but to use its generalized form π_θ as a good starting policy for acquiring the model on which to base an optimal policy. This is illustrated in the following procedure.

```
Ml-learning($\mathcal{D}, s$)
    extract $\pi_d$ from $\mathcal{D}$
    generalize $\pi_d$ into $\pi_\theta$ with supervised learning
    $\pi \leftarrow \pi_\theta$
    until Termination do
1       $a \leftarrow \pi(s)$
        perform action $a$ and observe resulting state $s'$
        update estimate of the model $\gamma$ and Pr
2       compute $V^\pi$ and $Q^\pi$
3       $\forall s, \pi'(s) \leftarrow \text{argmax}_a Q^\pi(s,a)$        // greedy policy for $V^\pi$
        $\pi \leftarrow \pi'$; $s \leftarrow s'$
```

Algorithm 10.7. Ml-learning, a model-based imitation learning procedure

The steps 2 and 3 of Ml-learning correspond to an incremental version of policy iteration (Algorithm 9.1). In PI, the actor starts with a full knowledge of the domain, from which it searches for an optimal policy. Here the actor/learner starts with π_θ, obtained from the demonstrated π_d. It progressively acquire the model and improve π. Note that step 1 may rely on Select to keep some level of exploration.

There is no clear cut of which is preferable, the model-free or the model-based approach. Several domain-dependent considerations interfere, taking into account that acting and learning are interleaved in RL. If a good simulator is available, then the model-based approach can start acting in the real world with a good estimate of the model and an improved policy. On the other hand, if the demonstrated policy π_d provides a good coverage of the domain, then the model-free approach can start with more confidence a Q-learning procedure.

The preceding approaches assume that the learner is given a reward function which is consistent with the teacher's demonstrations, i.e., the teacher illustrates how best to maximize the expected reward in the demonstrated cases. Otherwise, the learner may either start off track if the demonstrations are not good, or it may modify its policy far from what it is supposed to imitate if its reward function does not express what's the desired behavior. This consistency problem, well known in human interactions,[16] is avoided in inverse reinforcement learning.

10.7.3 Inverse RL

In inverse reinforcement learning (IRL), no *a priori* reward function is given to the learner. It is assumed that the reward function is reflected in the optimal behavior illustrated in the teacher's demonstrations. Instead of imitating the demonstrated policy, the learner seeks to acquire the underlying reward function that drives the teacher's behavior. The rational is to go beyond a simple imitation, since it is brittle and does not allow the learner to grasp what is sought. Learning what the teacher is trying to maximise, what motivates its actions, gives a better leverage for performing the task at hand in varying contexts.

IRL usually assumes that *(i)* a domain model is available, except for the reward function, and *(ii)* the demonstrated trajectories $\mathcal{D}$ provide the optimal policy $\pi^*(s)$ for all s in $\mathcal{D}$. A simple statement of the IRL problem is the following:

- find a function $r(s, a, s')$ such that $\forall s \in \mathcal{D}$, $\text{argmax}_a\, Q(s, a) = \pi^*(s)$, with
- $Q(s, a) = \sum_{s' \in \gamma(s,a)} \Pr(s'|s, a)[r(s, a, s') + \max_{a'}\{Q(s', a')\}]$ (Equation 10.3).

Even if $\mathcal{D}$ covers the entire state space, this formulation is under-specified. It has an infinite number of solutions. Beyond the equivalence of reward functions through shaping, many of the possible solutions have no interest. For example, an unvarying action-value function, $Q(s, a) = q(s)$ for all applicable a, meets to above specifications with a useless reward function.

This issue, common to all inverse problems, is generally addressed through optimization. For example, the learner may seek a reward that maximizes the "*sharpness*" of Q, i.e., the difference between $Q(s, \pi^*(s))$ and $Q(s, a)$ for any other action. The problem statement can be extended with

- maximize $\sum_s [Q(s, \pi^*(s)) - \max_{a \neq \pi^*(s)} Q(s, a)]$.

In a memory-based representation, this problem can be solved by linear programming.

As discussed earlier in this chapter, parametric representations are significantly more powerful for learning. In parametric IRL we define the reward as a function

[16]The meta-advice "*do as I say, but not as I do*" illustrates advisers' awareness of their inconsistencies.

r_θ, usually linear, of s and a, and seek to estimate its parameters. Let π_θ be a near optimal policy for the reward function r_θ, and π_d the demonstrated policy. A general schema for parametric IRL is the following:

```
IRL($S, A, \gamma, \Pr$)
1  define a function $r_\theta$ of $S$ and $A$ for some initial vector of parameters $\theta$
   until termination do
2      compute $\pi_\theta$, a near optimal policy for $r_\theta$
3      update $\theta$ to reduce the deviation of $\pi_\theta$ from $\pi_d$
```

Algorithm 10.8. IRL, a general schema for a parametric imitation learning.

Step 1 of IRL requires the definition of a vector of *features* $\boldsymbol{\phi} = [\phi_1, \dots, \phi_n]^\top$ that are characteristic functions of the states and actions which allow expressing parametrically the reward function as:

$$r_\theta(s, a, s') = \sum_{1 \le i \le n} \theta_i \phi_i(s, a, s') = \boldsymbol{\theta} \cdot \boldsymbol{\phi}(s, a, s')$$

Hence the designer does not need to provide r_θ, but she needs to specify meaningful features for its parametric decomposition. For example, in a robotics task involving navigation, a feature can be the Euclidian distance to a goal. Step 2 can use an incremental planning algorithm. The critical part in IRL is step 3. Comparing directly two policies is not meaningful: a difference in just one or a few states could have a critical impact for achieving the task. The idea is to estimate the deviation of π_θ from π_d from the difference between the two corresponding value functions V^{π_θ} and V^{π_d}, or the action-value functions Q^{π_θ} and Q^{π_d}. One approach, among many, decomposes V^π with respect to the features $\boldsymbol{\phi}$. Recall that:

$$\begin{aligned} V^\pi(s_{j_1}) &= \mathbb{E}[\sum_\sigma r_\theta(s, a, s')] \text{ over trajectories } \sigma_j = \langle (s_{j_1}, \pi(s_{j_1})), \dots, (s_{j_m}, \pi(s_{j_m})\rangle \\ &= \mathbb{E}[\sum_\sigma \sum_i \theta_i \phi_i(s, a, s')] \end{aligned}$$

Let the *expected value of the feature* ϕ_i in s for π be $\mu_i^\pi(s) = \mathbb{E}[\sum_\sigma \phi_i(s, \pi(s), s')]$. We can rewrite V^π as:

$$V^\pi(s) = \sum_{1 \le i \le n} \theta_i \mu_i^\pi(s) = \boldsymbol{\theta} \cdot \boldsymbol{\mu}^\pi(s)$$

Hence, we can assess the deviation between π_θ and π_d from the difference between μ^{π_θ} and μ^{π_d}. The latter is estimated from the demonstrated trajectories in $\mathcal{D}$ as:

$$\mu_i^{\pi_d}(s_{j_1}) = \frac{1}{N} \sum_{\sigma \in \mathcal{D}} \sum_k \phi(s_{j_k}, \pi_d(s_{j_k}), s_{j_{k+1}})$$

The computation of $\boldsymbol{\mu}^{\pi_\theta}$ can be integrated into step 2 since, as we saw in Chapter 9, V and π remain computationally related.

As the reader might guess from Algorithm 10.8, IRL is an offline computationally demanding task. Now, acquiring a good reward function r_θ can be amortized by further reinforcement learning and improving the actor's behavior for the task across the domain, in particular when this domain is not stationary. Note however that this IRL formulation requires a good initial knowledge of the domain (S, A, γ, Pr and the features $\boldsymbol{\phi}$) in addition to the teacher's demonstrations. Ongoing research is exploring model-free and deep learning for IRL in specific tasks.

10.8 Acting, Planning and Reinforcement Learning

Reinforcement learning is tightly linked to acting and planning. On the one hand, RL interleaves naturally acting and learning, at least in principle: this is explicit in the steps "perform action a" in previous RL procedures. On the other hand, MDP planning and RL have many commonalities. Both rely on the same representation and concepts of probabilistic state transition systems. Both seek a policy for achieving a task optimizing an expected utility or cumulative reward. The depart in a few ways:

- planning requires a predictive model of the domain, while RL learns a model from trial and error,
- planning can address any task in the domain, while learning acquires with a model specific to a given task.

However, in many practical settings, these differences are not substantial. We already mentioned that sampling complexity of RL is high and that trial and error in the real world is very costly, even unfeasible in critical domains. In practice, RL requires a simulator. A simulator is also needed for online MDP planning with a *generative sampling model* as given by the function Sample (see Definition 9.26). If we redefine the function Sample to returns pairs Sample$(s, a) = (s', r(s, a, s'))$, then it is straightforward to restate these MDP planning algorithms as maximizing expected cumulative rewards, instead of minimizing expected costs. We can also rephrase the steps in previous RL procedures saying "perform action a", and "observe resulting state and reward" into the step used in planning algorithm, that is: "$(s, r) \leftarrow$ Sample(s, a)". This would allow specifying a common algorithmic framework describing in a similar way the main steps of MDP planning and RL algorithms, e.g., for exploration, backup and update,

The purpose of this section however is not to develop these notions (briefly referenced in next section), but to discuss how a planner can be of help in RL and used in a continual online acting, planning and learning framework.

Planning for RL. Two ideas are of interest for coupling a planner with RL:

- Parametric RL allows generalizing, across high dimensional state and action spaces, what has been learned in some samples to neighboring cases.
- Monte Carlo sampling, MCTS and UCT algorithms allow for a receding horizon controllable and focused search that is very informative for a given state.

To keep the desirable generalization capability, let us rely on RL to acquire a parametric value function Q_θ that can be of use for planning and acting as well. Let us also rely on an MDP planner in two main steps of a parametric RL procedure such as Deep Q-learning:

- *the exploration strategy*: instead of a Select function choosing with a random ϵ-greedy (or Boltzman) distribution, a look-ahead search will rule out efficiently inappropriate options and focus the learning on the most promising ones.
- *the target value computation*: recall that the target y results from a one step maximization in Parametric Q-learning or Deep Q-learning, the latter requiring a target network $[\hat{Q}_{\theta^-}]$. A look-ahead planning implements such a maximization in a focused deeper search. It provides a more informed target.

These two steps should be coupled in a single call to a planner to guide the exploration and provide the corresponding target. At an early learning stage, the planning look-ahead can be extensive (large d in MCTS or UCT). Increased learning may progressively lead to shallower planning searches, until eventually no more search nor even learning are needed when the current Q_θ is systematically very close to the target (assuming a stationary domain).

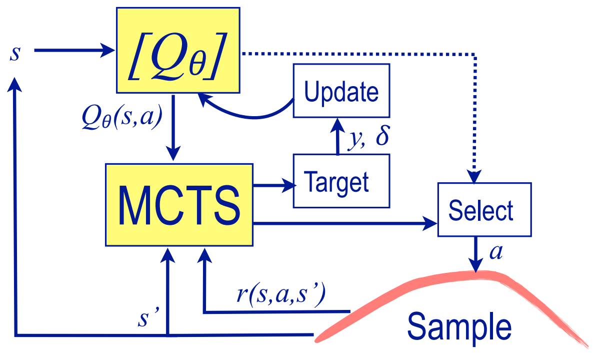


Figure 10.6. A schematic view of Deep Q-learning with an MCTS MDP planner replacing the target network, driving the exploration strategy, and interacting with a generative sampling model. The link from the neural net $[Q_\theta]$ to Select gets stronger when more is learned about the task.

The schematic view of Deep Q-learning in Figure 10.5 can be modified in Figure 10.6 with the introduction of an online sampling planner as MCTS, instead of the $[\hat{Q}_{\theta^-}]$ network. The expected benefit of replacing the target network with MCTS sampling is not in computational efficiency. It is mainly in a better learning exploration strategy and more informed target values. Both should speed-up learning. In this schema, the planner/learner interact with a generative sampling model. The approach is compatible with the use of replay-memory and minibatch buffers for back-propagation updates in $[Q_\theta]$. MCTS is initialized with the current Q_θ and uses the associated policy $\operatorname{argmax} Q_\theta$ to drive Rollout. The updates in MCTS (Line 3) remain local; only the final returned value $Q(s_r, \pi(s_r))$ affects Q_θ as the target y. We leave the specification of a detailed pseudo-code of this schema to the reader (see

Exercise 10.3).

Continual online acting and learning. Having learned with a generative sampling model a function Q_θ, we can use it to act in the real world. However, a simulator is never perfect. It can depart from the real world in many ways, e.g., Sample may ignore possible outcomes, follow a biased distribution, or return erroneous rewards. Furthermore, the real world is seldom stationary while the simulator is. It is better to keep improving what has been learned while acting.

There may be less incentives to keep planning at this stage. On the one hand, the actor is no longer in an exploration stage, it is mostly in an exploitation stage. On the other hand, we now seek to go beyond about what the generative sampling model gives, to which planning is limited.

In summary, the actor first uses a plan-and-learn schema with a simulator as in Figure 10.6. When it is confident enough, it moves to a continual act-and-learn schema as Figure 10.5. The latter should be adapted to online acting, i.e., multiple episodes starting from random states, replay-memory and mini-batch buffers are no longer needed. But possible discrepancies between estimates Q_θ and targets y have to be monitored to trigger more offline learning with, possibly an improved simulator.

10.9 Discussion and Bibliographic Notes

In this section we discuss reinforcement learning techniques presented this chapter, refer to their sources and additional material. A broad view of RL can be found in e.g., the book of [1070] or the surveys [569, 1161].

Foundations. The study of reinforcement learning started well before AI in the mid-nineteenth century with early work in animal training and psychology. Examples are the well-known work of Pavlov about conditioned dog reflexes with reinforcement stimulus, or the work of Thorndike who proposed a *"law of effect"* of reinforcing events on animals actions [1095]. These early contributions where extensively developed in experimental and behavioral psychology, e.g., with Skinner's "principle of reinforcement" and his apparatus to measure a reinforcement strength [1032].

The principles of punishments and rewards for intelligent machines have been mentioned by Turing [1108]. Early implementation of trial and error approaches go back to 'cybernetic turtles' of Shannon [1000] for the exploration of mazes. Shannon considered learning capabilities for his chess player [999]. These capabilities where developed by Samuel [976] as an early temporal difference update method with a parametric approximation in his checker player, the first program that was able to learn enough to defeat its designer.

RL and MDP. The modern formulation of RL in an MDP framework goes back to the seventies with contributions from Werbos [1167], Sutton [1068], Barto et al. [98] and others. A theoretical formalisation is developed in Bertsekas and Tsitsiklis [131].

There are many links between MDP planning and reinforcement learning. This is in particular the case for Monte Carlo planning techniques (Section 9.5.5) and RL. Their links, discussed [1143], have been illustrated in several developments, e.g., the Texplore MCTS-based system [499]. These links are also strong between planning and model-based RL. The latter estimates and progressively improves the underlying MDP model of domain [799]. A common algorithmic framework giving jointly the main steps of planning and RL algorithms has been proposed in [798].

The classical task-oriented RL formulation can be extended to a goal-oriented formulation. Given a set of goal states S_g, the action-value function is conditioned on the goal state: $Q(s, a, s_g)$ for $s_g \sim S_g$. The update rule of Equation 10.4 in Q-learning with this Q function optimizes the total expected reward over policies reaching a goal. This approach has been successfully tested in several works, e.g., [565] (which minimizes the expected sum of actions costs), or [984, 43, 456] in various other settings.

Most RL formalizations introduce a discount factor, which is needed for summing up over an infinite horizon. In this chapter, as well as in the two previous ones, we do not need to introduce a discount factor. We take the task achievement or goal reachability view instead of the process maintenance view (see Section 8.3.1). Tasks have a finite number of steps; too long tasks need to be broken down into short subtasks for RL to converge; unlimited loops need to be detected and stoped as failures. Furthermore, the practical setting of a value for a discount factor is problematic. Its value changes significantly the solution [569]. For a robotics RL problem, a small discount leads to unstable control [622]. Taking a discount close to 1 is equivalent to optimizing for the expected average-reward, i.e., $\lim_{h\to\infty} E[1/h \sum_i r_i]$.

Value-Based RL. The Q-learning algorithm studied in this chapter was proposed by Watkins and Dayan [1158]. SARSA (for State, Action, Reward, State, Action) is a similar value-based RL [964]. It takes into account a sequence of two steps (s, a, s', a') before performing the update of the estimated quality of a in s. The update term of Q in Equation 10.4 is no longer with $\max_{a'}\{Q(s', a')\}$ but with $Q(s', a')$ for the actually taken action a' in s'.

Other value-based RL proceed by updating $V(s)$ rather then $Q(s, a)$. Updates are performed over tuples (s, a, s') in a similar way: $V(s) \leftarrow V(s) + \alpha[r(s, a, s') + V(s') - V(s)]$. This algorithm called $TD(0)$, is part of a family of algorithms $TD(\lambda)$ which perform updates over previously visited states, with a decreasing weight depending on the frequency of meeting each state [1092].

Let us also mention the DYNA algorithm and its variants that combine model-based and model-free RL [1069]. DYNA maintains and updates an estimate of transition probabilities $P(s'|s, a)$ and rewards $r(s, a, s')$. At each step two updates are performed, a *Q-learning* type with $Q(s, a) \leftarrow \sum_{s'} P(s'|s, a)[r(s, a, s') + \max_{a'}\{Q(s', a')\}]$, for the current s and a, and a Value Iteration type for other pairs (state, action) chosen randomly or according to certain priority rules, taking into account new estimates. Here, the experience allows estimating the model and the current policy. The estimated model in turn allows improving the policy. Each step is more computationally expensive than in *Q-learning*, but the convergence occurs more rapidly in the number of update steps.

Parametric value-based RL appeared very early using various methods. A good illustration is a backgammon player that defeated the human world champion [1091]. Several contributions developed parametric RL in continuous state spaces, e.g., [1072]. Recently, neural nets became dominant in parametric RL.

Neural Nets and Deep RL. Neural networks go back to the pioneering work of McCulloch and Pitts [772]. Since the Deep NN successes of the early 2010 in image classification, the field has extended over a huge literature, the history of which is beyond the scope of this section. The AI textbook of [967] gives a good introductory coverage of deep learning. There are several specialized technical books about neural nets, including those of [438], [217] or [12]. Numerous tutorials, online courses, tools, and cloud-based platforms are available, with a main focus on image, speech and natural language applications.

Deep neural nets have been widely adopted in RL since 2010. DQN was developed in [797] and improved in e.g., in [449]. Several analyses of DQN mechanisms such as memory reply and target nets have been undertaken, e.g., in [9, 338]. Stochastic mini-batches selection methods such as prioritized sweeping techniques are discussed in [806, 41, 1154]. Batch normalization methods have been proposed to avoid data distribution variations [541]. The RMSProp learning rate control method is discussed in [606]. Dropout techniques, i.e., temporarily removing random units from the network, have been developed to address overfitting problems [1050]. Other deep RL techniques are covered in these surveys [53, 458, 688].

Parametrized RL approaches with DNN offer powerful generalization capabilities. They suffer however from several drawbacks, among which opacity, brittleness, and sampling complexity. A broad set of very active investigations are addressing these issues. Let us mention in particular the *neurosymbolic* approaches. These seek to merge prior explicit knowledge, in various differentiable representations, with NN parametric approximation techniques. The purpose is to leverage the added knowledge for extending the learning domain, guiding the learning, reducing drastically the sampling complexity, and being able to explain, verify and prove the learned behavior. Logic-based neurosymbolic approaches are illustrated with techniques such as logic tensor networks [995, 74], neural logic machines [303], and several other approaches, e.g., [383, 38]. *Programmatic* reinforcement learning methods explore a programming-based neurosymbolic approach [1131, 1133, 923]. They use a domain specific programming language to express high-level constraints on the policy to be learned. A key issue is how to approximate this knowledge into a differentiable program amenable to NN learning.

Among the variety of NN architectures, some are well adapted to a uniform and systematic organization, e.g., convolution nets for images [645], or Recurrent nets for temporal sequences in natural language processing. A few other architectures appear to be more adapted for handling a collections of relations, as typically used in a structured state representation. Among the latter, notably *Graph Neural Networks* (GNNs) operate directly on graph-structured data. A GNN is a network whose cells are organized as the graph or hypergraph of a set of relations. Learning with a GNN can be about the vertices, edges, or the graph structure [462]. RL with GNN has been tried in a variety of approaches, e.g., [1052, 441, 470, 1152].

Finally, let us mention that the task of choosing and tuning NN architectures for a given learning problem can also be learned with NN. This is the idea of *Differentiable Architecture Search* (DARTS) [727, 1220], which has been mostly demonstrated in image and langage processing, but was also found relevant for Programmatic Reinforcement Learning and aided RL.

Policy-Based RL. Policy-based approaches have been widely studied for learning, particularly in robotics and control applications, for their ability to cope with high dimensional continuous action spaces. The policy gradient idea is due to [1176]. A fundamental result is the "*stochastic policy gradient theorem*" of [1071], which expressed the gradient ∇V^{π} as a fonction of $\nabla \pi$ and ∇Q, and also stated the compatibility conditions for unbiased action-value estimates. Convergence, optimality and other properties of policy-gradient algorithms have been studied [1176, 1071, 889, 135].

RL with parametric policy iteration methods where developed with a variety of parametrization approaches, e.g., least squares temporal difference methods [666], or random forest classifiers [667].

The actor/critic architecture received many contributions, such as [634, 890, 136, 287]. These contributions, as most of the dominant methods in policy-based approaches, relied for a while on *stochastic policies*, which map a state to a probability distribution over the action space. Stochastic policies are appealing for their exploration capability and the smooth effects of parameter updates on action values. However, the stochastic gradient integrates over the state space as well as the action space. The latter is not needed in the deterministic policy gradient case. Hence, stochastic policy gradient RL requires extensive trials and is less efficient than the deterministic case.

Deterministic policy gradient approaches gained favor with [1017], which proved an equivalent and simpler deterministic policy gradient theorem and the corresponding compatibility conditions. This paper also proposed and empirically tested several actor/critic on-policy and off-policy methods, among which the COPDAC-GQ algorithm which combines compatible updates with a gradient Q-learning critic. A theoretical analysis of DPG is developed in [1190], which establishes its convergence rate toward a near optimal policy under different conditions.

Several additional techniques have been proposed to improve and accelerate policy-gradient methods, such as importance sampling [700], the use of the so-called 'natural gradient' which provides steeper updates [571, 890, 796], constraints for guiding the policy optimization [990], or their variants with an objective function that enables multiple epochs of minibatch updates [991]. Gradient-free random search alternatives such as [1073] have been proposed Some of these and other policy-base RL techniques are surveyed in [53, 448] and, more specifically for robotics, in [622, 288].

Hierarchical and relational RL. When the task to learn involves long sequences of primitive actions, the sampling complexity of RL becomes too high, and makes learning hardly tractable. A natural idea is to try breaking long sequences into a hierarchy of shorter ones corresponding to a hierarchy of tasks and subtasks. Hierarchical RL (HRL) has attracted a large amount of contributions, surveyed in [871].

Most work in HRL considers the task hierarchy to be given as input. Among these, the MAXQ is an interesting approach [297] which proceeds by decomposing the Q-value function into separate functions for the subtasks. A subtask may use primitive actions or other subtasks. MAXQ seeks optimal policies for the subtasks, but not for overall hierarchical policy. An alternative approach relies on a hierarchy of stochastic finite state-machines [39, 75]. It generalizes "*Partial programming*" techniques, where specified partial programs leave out unspecified steps to be learned by reinforcement [870].

A similar idea is explored in the "*Policy sketches*" idea [42], which seeks to learn composable policies for multiple tasks that have common subtasks. The learner has access to a domain simulator and is given a high-level abstract sketch for each task as a finite sequence of symbols $\langle b_1, b_2, \ldots \rangle$. A symbol b refers to a subtask that will be achieved with a policy π_b to be learned, which will be common to all tasks sharing b. Learning uses an actor/critic approach together with a curriculum learning guidance. It has been evaluated on a few synthetic domains such as a 2D Minecraft game.

A few approaches do not take the task hierarchy as input. Some seek to synthesize it, e.g., the MAXQ hierarchy [781]. Others seek a unified learning method that jointly learns the subtasks and the associated hierarchical policy. Graph-based techniques are discussed in [783]. A broader coverage is given in the survey [871].

Relational RL is akin to hierarchical RL; it relies on structured or factored MDP (Section 8.2). It uses a relational representation to exploit background knowledge and ease generalization. It is related to learning Dynamic Bayes nets, for which there is an extensive literature, e.g., [960, 1146]. For example, [1060] learns a DBN structure and parameters as part of the RL procedure. An overview of Relational RL is given in [1074]. Relational RL with deep neural nets led to a few developments, e.g., [1219] which demonstrated its results in the StarCraft video game.

Transfer and Multitask Learning. As said earlier, RL focuses on a single task in a single domain. Generalizing what has been learned for a given task and environment to other similar tasks and environments has been studied by psychologists for decades; it remains today a significant challenge in RL.

In the narrowest sense, transfer learning seeks to extend the learned model to 'real data', which may differ from the distribution of the training data. This is typically an issue for data classification tasks, dominant in the transfer learning literature. A related issue is the "*sim-to-real*" transfer, of interest in particular in robotics or autonomous driving, e.g., [867, 89, 711], and the survey [1228]).

More specifically for RL, transfer learning is about leveraging a model learned for a source task in order to more efficiently learn other target tasks. A simple idea in parametric RL would start learning a model for, e.g., a video game, from a parameter distribution learned in another similar game, instead of initializing the parameters with a random distribution. One may also use the source model to drive the exploration for learning the target task.

Multitask learning addresses a similar concern from a different perspective: the learner is initially confronted with several tasks in the domain and handles them jointly. A hierarchical Bayesian approach is illustrated in [1177]. The frontier with transfer RL is not crisp; several techniques can be used for both.

Numerous elaborate multitask and transfer RL methods have been proposed for various cases of how the target differs from the source task. When the source and target are within the same state and action spaces and vary only in their goals and reward functions, transfer may rely for example on policy reuse or reward shaping. It is made easier with hierarchical RL. One may also use the MCTS planning/RL combination, as in Section 10.8, to explore with the source knowledge learning the target model.

Transfer across distinct domains is more complex. When the actions spaces are dissimilar, mappings from the source's actions to the target's are sought, e.g., [1083]. Different state spaces require finding invariants across states, or abstracting states, as in the meta-RL approaches. Good examples are Meta-Q learning [337], AdaRL [526], and RoML [445]. An alternative relies on logic tensor networks to leverage prior knowledge across domains [73]. Other relavant approaches are [746, 936, 140, 693]

A comprehensive summary of this very broad area of transfer and multitask learning is beyond the scope of this brief discussion. We refer the reader to several surveys, such as, for a global view on transfer [1100], for a map of transfer methods in RL [1082, 687], and [1231] transfer with neural net techniques.

Aided RL. Section 10.7 illustrates a few among the many approaches that seek to give a feedback to a learner about desirable behaviors, to reduce the sampling complexity of RL and improve learning performance with additional information from a human. Let us discuss some of these approaches here.

Techniques for overcoming sparse rewards that hinder many applications, particularly in goal reachability domains, include shaping, credit assignment, transfer learning, and a variety of heuristics. The reward shaping property is due to [847]. Many contributions to heuristics for RL have been proposed, with or without shaping, e.g., [138, 139, 228, 394]. Reward shaping with natural language input has been initially investigated in [442]; it is further discussed in Section 23.2.3.

Credit assignment is about rewarding actions that may not achieve the goal but contribute to reaching it. Supervised learning and other approaches are explored for estimating these assignments, e.g., [473, 55, 1106, 51], including for the purpose of transfer learning [355].

RL techniques leveraging aids from a teacher includes reinforcement learning from demonstrations (RLfD), apprenticeship and learning from advices, imitation learning, and inverse RL. These categories of methods have significant overlaps. RLfD is a very active area of investigation, particularly in robotics. When demonstrations take place physically and are performed directly by a person (as opposed for example to teleoperated demonstrations), the learner has first to perceive and interpret what the teacher is doing, which may lead to activity and plan recognition issues, then it has to map the observed actions from the teacher sensori-motor space to the learner's, and to generalize what it has been demonstrated. The following contributions [636, 851, 1121] and surveys [50, 938] illustrate some approaches. A different focus arises when demonstrations are assumed to be given in an abstract form, as the set $\mathcal{D}$ of Section 10.7.2, or as a teacher's interactions with a computer. For example, a Q-learning from demonstration algorithm called DQfD is proposed in [500]; a shaping approach to RLfD is developed in [186]; a state-space abstraction approach is considered in [245]; while [1226] relies on Bayes nets extracted from demonstrations.

Imitation learning is closely related to RLfD. It also relies on demonstrations, but often in a narrower sense. It seeks to map the teacher's behavior into a policy, used afterward mainly in a greedy exploitation mode, without much exploration. An overview of imitation learning is proposed in [61]. Imitation policies tend to be brittle and drift over long term. These drawbacks are addressed for example with SQIL, a Q-learning approach which seeks to stay close to demonstrated states [939], or with Propel, an imitation programmatic RL relying on prior domain knowledge [1132].

An alternative is to learn the reward function that drives the teacher's behavior instead of the resulting policy. It is generally more robust to imitate the teacher's motivations then its policy. This is inverse reinforcement learning (IRL), which, like imitation learning, is akin to inverse optimal control. However, while imitation learning maps demonstrated trajectories to a policy, IRL maps them to a reward function. IRL was proposed by [965], a broad historical perspective from inverse optimal control to IRL is given in [2]. Contributions to IRL include [848, 3, 838, 839, 361], and others to be discussed next (paragraph 10.9). Several IRL methods are surveyed in [52].

IRL learns first a reward function, then uses that function to learn how to act. Generative Adversarial Imitation Learning (GAIL) aims at merging the two learning stages into a single one [503].[17] It uses a maximum entropy IRL method [1235] with an approach akin to *Generative Adversarial Nets* [439].

Learning from advices fit into a broad class of interactive or "human-in-the-loop" RL. Contributions related to interactive RL methods include systems such as TAMER [1156], COACH [54], and others [745, 1224]. Some approaches demonstrate learning from sparse human feedback without access to a reward function, e.g., from preferences about trajectories to the goal [233], from supervision feedback [1159], or guidance in natural language [741]. A broad set of interactive RL techniques are surveyed in [828, 56].

Finally, let us mention approaches (discussed in Chapter 16) that help the learner by organizing the learning task, as with developing a learning curricula.

Offline RL. RL algorithms are fundamentally online: they interact through trial and error with either a domain simulator or through real experiments. The latter are often too costly or risky. In some cases, simulators are unavailable and cannot be developed, but plenty of data about the outcome of past actions is accessible. This holds for e.g., most health care applications. Offline RL seeks to address these cases for learning a behavior in a purely *data-driven* way, without exploration nor interaction with the domain.

[17] see also: https://uscresl.github.io/humanoid-gail/

For small problems with low dimensional data, *batch RL* approaches apply successfully the usual off-policy RL algorithms to a training dataset $\mathcal{D} = \{(s, a, r, s')\}$ of comprehensive samples [679]. But since exploration is impossible, offline RL cannot succeed if the training set $\mathcal{D}$ does not cover well enough high-reward and high-risk regions of the domain; which is a real challenge for complex domains. Moreover, batch RL approaches suffer from a *distributional shift* issue: one learns a different policy, hopefully better, then the training behavior in $\mathcal{D}$. This shift about of how much the learned policy differ from the training one. Different approaches, such as off-policy importance sampling or regularization of the Q function, are being explored to handle offline RL issues. A detailed survey is given in [701].

Continual learning. The learner in CL updates its model from a continuous stream of information throughout time such as to adapt to a changing environment. CL is very important in nonstationary domains. As most parametric ML techniques, it suffers from "catastrophic forgetting". The latter refers to parameter updates that erase part of previous knowledge. Forgetting can be mitigated with various rehearsal and replay methods [1112, 45], and methods inspired from transfer learning [457]. A comprehensive survey of CL issue is given in [869].

RL applications in games. Despite several celebrated success stories and impressive lab demonstrations, RL is not as widely deployed as, for example, supervised learning in image, speech and language applications. This is due to the high sampling complexity of most RL methods and, consequently, to the necessity of a simulator or an equivalent generative sampling function. Because of this last requirement, RL has been mostly applied in games, and in robotics, often for control tasks in motion, manipulation and navigation.

We already mentioned the early TD-Gammon backgammon player [1091]. On video games, the DQN system [797] successfully demonstrated the level of a professional human player on most Atari games. An exception with low performance is the 'Montezuma's revenge', a complex and huge state space labyrinth exploration game with treasures, enemies, traps, etc. This game required elaborate exploration capabilities which were addressed by systems such as h-DQN [652], Go-Explore [318], or DeepSynth [474]. Dota 2 and Starcraft are two even more challenging video games with a huge action spaces, partial observability and a long horizon play; they were successfully addressed with self play RL by respectively OpenFive [122] and AlphaStar [1142], the latter combining self play with imitation learning.

For board games, Deep RL methods reached fame with the AlphaGo system [1018] by beating in 2016 and 2017 the world best go professional players. AlphaGo was subsequently improved by its designers as AlphaGo Zero [1019], then AlphaZero [1020]. The latter demonstrated superior learning performances also in chess and in shogi, a Japanese variant of chess. The RL method in AlphaZero relies on self-play training with MCTS. However, instead of performing a rollout to assess a position in a leaf s of the MCTS tree (step 3 in Algorithm 9.26), it uses a neural net to estimates for this leaf the expected value function $V_\theta(s)$, as well as a stochastic policy distribution $\pi_\theta(s)$. The leaf s is developed with a move a that combines $V_\theta(s)$ (averaged over follow-up positions), $\pi_\theta(s)$ and less frequently tried moves (i.e., $n(s,a)$ in MCTS). The MCTS search returns to the root node s_r a stochastic policy π which is used to select and play a move in s_r, either greedily or with exploration (using $n(s_r,a)$). The neural net is trained on self-play with the reward obtained on a terminal state. The network parameters are updated to minimize the error between the predicted V_θ and the game outcome, and to maximize the similarity of the estimated π_θ to the MCTS policy π (actually, the loss function in AlphaZero is the sum of the mean-squared error for V and the cross-entropy loss for π; conceptually, one may view the NN part as two networks estimating respectively the value function and the policy).

RL has been applied successfully to other types of games, such as:

- puzzles, usually solved with search methods but for which RL can be competitive, as in the Rubik's cube, addressed with an approach combining MCTS with a joint value–policy network [769];
- chance card games such as poker, for which a multiplayer system called Pluribus has defeated professional players relying on a strategy learned offline with self-play, and refined online [181].

RL applications in robotics. The development of RL in robotics responds to strong practical needs. It alleviates the hard task of robot programming. Furthermore, prior knowledge of adequate robot movements is seldom available as such; it requires elaborate modeling, planning and optimization (see Part VII). RL in Robotics started in the eighties and early nineties with low-level cart balancing or box pushing tasks, e.g., [748]. Numerous robotics RL developments have been proposed, often tested on classical benchmarks such as the cartpole, inverted pendulum, unicycle or bicycle control problems [653, 862]. These developments, further detailed in Chapter 22, are discussed in several surveys, among which [622, 643, 910, 195, 600].

Application-wise, RL has been demonstrated in robotics in many tasks, e.g.,

- outdoor navigation in complex terrain with a tight coupling of perception and motion [1016];
- wheelchair path planning in a social environment [605];
- bipedal walking [711, 314] and control [691], quadrupedal walking [1038] and navigation [963];
- helicopter aerobatics flying, a very difficult task for human pilots, [4, 244, 6];
- dexterous manipulation [216]; manipulating and solving the Rubik's cube with the finger movements of a single hand [21];
- autonomous driving in various configurations, see the survey [608].

Often, these developments relied on elaborate simulators, such as MuJoCo [1098]. They required extensive training (e.g., several runtime months for the dexterous Rubik's cube manipulation), and sim-to-real transfer.

In conclusion, learning from simulation, self play and trial and error experiments makes RL very appealing, although very costly in sampling complexity, and computational expenses.[18] RL is appealing because it spares the hard task of developing

[18]The training of the Rubik's cube dexterous manipulation consumed about 2.8 GWh of energy, corre-

and testing formal domain models. But this has drawback. Indeed, while possible errors and failures in game applications may be inconsequential, other areas, like most robotics applications have strong safety requirements. They demand for the deployment of validation, verification and certification methods, as well as explanation capabilities for a trustful use. For that, formal models are needed. Significant research is still required for their efficient integration to RL approaches.

10.10 Exercises

10.1. Detail the computation of the partial derivative $\frac{\partial \|\tilde{y}-y\|^2}{\partial \theta_{i,j}^{L-1}}$ for a parameter $\theta_{i,j}^{L-1}$ in row $L-1$. Derive an update rule for this parameter, equivalent to Equation 10.14. Show how the proposed definition of δ^L and the update of Θ^{L-1} are entailed from this computation.

10.2. The loss function $\|\tilde{\mathbf{y}} - \mathbf{y}\|^2$ considered in Section 10.4 is very convenient for regression problems. For classification problems, the target $\mathbf{y}$ is a "one-hot" vector, i.e., a vector of zeros except for its i^{th} component equal to 1 for training instances of the i^{th} class. In that case, a more appropriate loss function is the *cross entropy* measure of the error between a predicted probability $\tilde{\mathbf{y}}$ and the label which represents the actual class given by $\mathbf{y}$. The cross entropy is defined as the dot product $Loss(f_\theta) = -\mathbf{y} \cdot \log \tilde{\mathbf{y}}$. Compute the partial derivatives and revise the Backpropagation pseudo-code for this loss function.

10.3. Develop the schema of Figure 10.6 into a detailed pseudo-code relying on the Deep Q-learning and UCT procedures.

sponding to about 1000 tons of CO2, the equivalent of a european household consumption over 7000 months. The OpenFive training for the Dota 2 game required about the double energy. Hopefully these figures should be significantly decreasing in the futur with more sampling efficient methods and better hardware [875]

Part IV

Nondeterministic Models

We see that in the course of time, the most unexpected things will happen. Seas will displace themselves, mountains will collapse, and some huge river will pour its water into the desert.

Lucretius, *De Rerum Natura*, 1st century BCE, likely between 58 and 55 BCE

Nondeterministic models, like probabilistic models (see Part III), drop the assumption that an action applied in a state leads to only one state. The main difference with probabilistic models is that nondeterministic models do not have information about the probability distribution of transitions. In spite of this, the main motivation for acting, planning, and learning using nondeterministic models is the same as those of probabilistic approaches, namely, the need to model uncertainty: Most often, the future is never entirely predictable without uncertainty.

Nondeterministic models might be considered as a special case of probabilistic models with a uniform probability distribution. This is not the case. In nondeterministic models we do not know that the probability distribution is uniform, we simply do not have any information about such distribution. This is indeed a main conceptual difference, and there are some main reasons to choose a nondeterministic model rather than a probabilistic one:

- In several cases we do not have information about the probability of an outcome. Unless we have some detailed and extensive statistics about the result of action applications, assigning probabilities to outcomes can be difficult and, in some cases, even misleading. Most of the probabilistic approaches tend to reduce the planning problem to an optimization problem that depends on the probability distribution of action transitions (see Chapter 9). If we do not have enough statistical data that provides an evidence of such distribution, planning with probabilities can produce misleading results.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- Similarly, reinforcement learning reduces the learning problem to an optimization problem that highly depends on how rewards are assigned, a task that in many cases is far from obvious and natural, and whose difficulty is often underestimated. If we do not have enough information and details about rewards, reinforcement learning can produce misleading results.
- In safety-critical applications, even outcomes with very low probabilities to occur must be considered, and they have the same importance as highly probable ones. For instance, even if a failure in the energy distribution of an airplane has low probability to occur, it is a key requirement do deal with such event. Even if a failure in bank transaction has low probability to occur, we must act, plan, and learn taking such failure into account.
- In many cases, our focus is not on finding an optimal policy with respect to some utility function. Instead, we are concerned with satisfying certain criteria, such as ensuring specific behaviors across the system. Traditional approaches based on Value Iteration or Policy Iteration (see Section 8.1.3) aim to minimize the cost of reaching a goal. However, in many applications, minimizing cost may not be the primary objective. Instead, the solution may need to satisfy certain logical properties. For example, we may prioritize a solution that guarantees safety, even if achieving this comes at a higher cost.
- In probabilistic approaches, safe solutions (safe policies in Section 8.1.2) may not represent satisfactorily safety requirements. Indeed, a safe policy is defined as the policy that has probability one to reach the goal (Definition 8.6). This definition does not take into account interesting differences that such safe policies can have. For instance, suppose we have a policy such that the application of an action to a state either results nondeterministically in the same state (no progress at all), or leads to the goal from the given state. This policy has probability one to reach the goal. It has the same probability of a different policy that applies a different action that leads to states from which we are guaranteed to reach the goal. However, while both policies have the same probability to reach the goal, the two solutions are very different, since the former involves a possible loop (and allows the possibility to get stuck in such loop), while the second does not.

These main conceptual differences lead to important practical and technical differences in acting, planning, and learning. For instance, we can use specific and effective techniques for acting, such as automata, Behavior Trees (BTs), or Petri Nets (PNs)(Chapter 11). We can address the planning problem (Chapter 12) with techniques (see, e.g., Section 12.3) that factorize the different possible outcomes in a compact representation of the set of possible outcomes, thus allowing us to deal with very large state spaces. This is possible since we do not have to take into account probabilities. In learning (see, e.g., Chapter 13), we can devise techniques that learn action schema by extending the approach presented in Chapter 4.

In this part of the book, we explore various approaches to using nondeterministic models to handle the uncertainty and nondeterminism in acting, planning and learning:

- The following chapter (Chapter 11) describes some main different representations of nondeterministic models and how they can be used for acting: Nondeterministic state transition systems and policies (Section 11.1), automata (Section 11.2), Behavior Trees (Section 11.3), and Petri Nets (Section 11.4).

- Chapter 12 is devoted to planning techniques with nondetermistic domains: And/Or graph search (Section 12.1), planning based on on determinization techniques (Section 12.2), planning via symbolic model checking (Section 12.3), planning by synthesis of input/output automata (Section 12.4), techniques for behavior-tree generation (Section 12.5).
- Finally, Chapter 13 is dedicated to learning with nondeterministic models. We describe some open challenges in learning action schema for nondeterministic models.

11 Acting with Nondeterministic Models

In this chapter we introduce different representations and techniques for acting with nondeterministic models: Nondeterministic state transition systems (Section 11.1), automata (Section 11.2), Behavior Trees (Section 11.3), and Petri Nets (Section 11.4).

11.1 State Transition Systems

Nondeterministic state transition systems allow for actions that lead from one state to one of many possible different states, representing in this way the uncertainty in the actual outcome of action applications. In this section, we recall the notion of policy that maps states to actions as defined in Part III for the probabilistic approaches. We define the notion of unsafe, safe cyclic, and safe acyclic policies depending of whether policies guarantee to lead to a goal state with different strength (e.g., just in some cases or in all possible cases). Acting with policies is a simple loop that observes the current state and applies the action specified by the policy for that state. Policies allow for acting in a conditional way depending on the actual action outcome and to repeat the application of actions in some states until the action leads to a different state.

Nondeterministic state transition systems relax the assumption that $\gamma(s, a)$ returns a single state. Then for every state s and action a, either $\gamma(s, a) = \varnothing$ (i.e., the action is not applicable) or $\gamma(s, a)$ is the set of states that may result from the application of a to the state s, that is, $\gamma : S \times A \rightarrow 2^S$.

A nondeterministic state transition system can therefore be described in terms of a finite set of states S, a finite set of actions A, and a transition function $\gamma(s, a)$ that maps each state s and action a into a set of states.A *nondeterministic state transition system* Σ is the tuple (S, A, γ), where S is the finite set of states, A is the finite set of actions, and $\gamma : S \times A \rightarrow 2^S$ is the state transition function. An action $a \in A$ is applicable in state $s \in S$ if and only if $\gamma(s, a) \neq \varnothing$. *Applicable*$(s)$ is the set of actions applicable in state s.

Example 11.1. In Figure 11.1, we show a simple example of nondeterministic model, inspired by a management facility for a harbor, where an item (e.g., a container, a car) is unloaded from the ship, stored in some storage area, possibly moved to transit areas while waiting to be parked, and delivered to gates where it is loaded on trucks. In this simple example, we have just one state variable, pos(item), which can range over nine values: on_ship, at_harbor, parking1, parking2, transit1, transit2, transit3, gate1, and gate2. For simplicity, we label each state in Figure 11.1 only with the value of the variable pos(item).

In this example, we have just five actions. Two of them are deterministic, unload and back, and three are nondeterministic, park, move, and deliver. Action unload

Free pre-publication, for personal use only. To be published by Cambridge University Press.

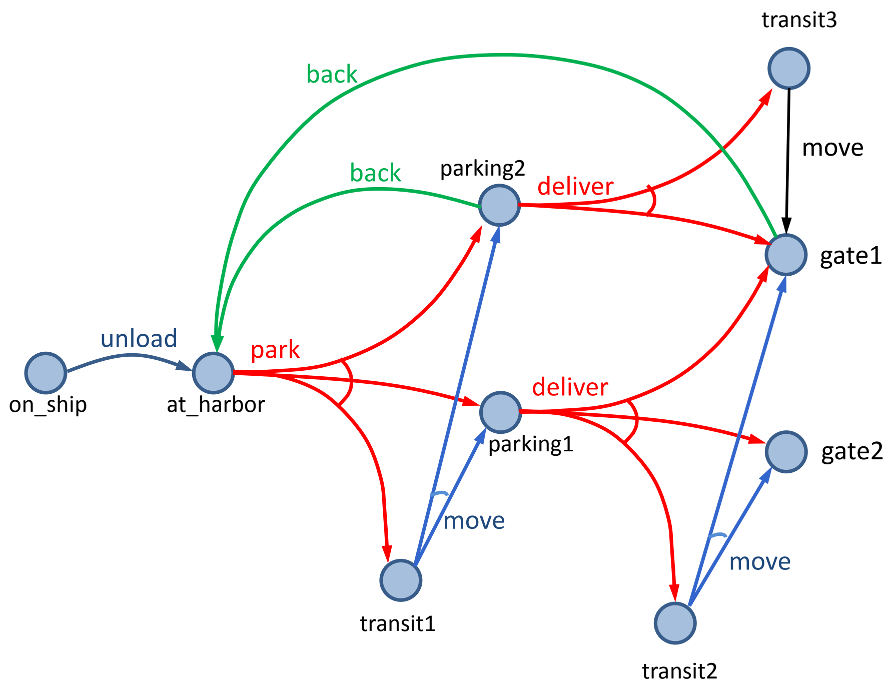


Figure 11.1. A simple nondeterministic planning domain model.

unloads the item from the ship to the harbor, its preconditions are pos(item) = on_ship, and its effects pos(item) ← at_harbor. Action back moves the item back from any position in the harbor to the position pos(item) = at_harbor. To keep the figure simple, in Figure 11.1 we show only two instances of actions back from the state where pos(item) = parking2 and the state where pos(item) = gate1, but a back arrow should be drawn from each state where the position is parking1, parking2, transit1, transit2, transit3, gate1, and gate2.

The actions park, move, and deliver are nondeterministic. In the case of action park, we represent with nondetermism the fact that the storage areas parking1 and parking2 may be unavailable for storing items, for example, because they may be closed or full. Whether an area is available or not cannot be predicted, because there are other actors parking and delivering items, for example from different ships. However, we assume that it is always possible either to park the item in one of the two parking areas or to move it to transit area transit1. The item waits in transit1 until one of the two parking areas are available, and it can be stored by the action move. Also in the case of move, we use nondeterminism to represent the fact that we do not know *a priori* which one of the two areas may become available.[1] From the two parking areas, it is possible to deliver the container and load them on trucks or to a transit area, from which it is necessary to move the container into either one of the two parking areas. The deliver

[1]In general, if an action's outcome depends on something that is unknown to the actor, then it is sometimes useful for the actor to think of the possible outcomes as nondeterministic. As an analogy, we think of random number generators as having nondeterministic outcomes, even though many of these generators are deterministic.

action moves containers from parking1 to one of the two gates where trucks are loaded or to a transit area from which it is necessary to move the container again to load trucks in one of the two gates.[2] The same action from parking2 may lead to gate1 or to another transit area. □

11.1.1 Acting with Policies

We recall the definition of *policy* in Section 8.1. A policy is a partial function that maps states into actions. Intuitively, if $\pi(s) = a$, it means that we should perform action a in state s. Let $\Sigma = (S, A, \gamma)$ be a nondeterministic model. Let $S' \subseteq S$. A policy π for a nondeterministic model Σ is a function $\pi : S' \rightarrow A$ such that, for every $s \in S'$, $\pi(s) \in Applicable(s)$. It follows that $Domain(\pi) = S'$.

We call our policies *memoryless policies*. A policy with memory is a mapping from a history of states to an action. Policies with memory allow for performing different actions in the same state, depending on the states visited so far. Moreover, our policies are (partial) functions, i.e. we restrict to *deterministic policies* that map a state to a single action. *Nondeterministic policies*, i.e., policies that map a state to more than one action, or *probabilistic policies* that map a state to a probability distribution over actions, can be useful in case we may need to select one among many different actions depending on. e.g., given, constraints. See, e.g., [291, 292].

Example 11.2. In the nondeterministic model of Example 11.1 shown in Figure 11.1, let π_1, π_2, and π_3 be the following policies:[3]

$$\pi_1 = \{(\text{on\_ship}, \text{unload}), (\text{at\_harbor}, \text{park}), (\text{parking1}, \text{deliver})\}$$

$$\begin{aligned}\pi_2 = \{&(\text{on\_ship}, \text{unload}), (\text{at\_harbor}, \text{park}), (\text{parking1}, \text{deliver}),\\ &(\text{transit1}, \text{move}), (\text{transit2}, \text{move}), (\text{parking2}, \text{back}), (\text{gate1}, \text{back})\}\end{aligned}$$

$$\begin{aligned}\pi_3 = \{&(\text{on\_ship}, \text{unload}), (\text{at\_harbor}, \text{park}), (\text{parking1}, \text{deliver}),\\ &(\text{transit1}, \text{move}), (\text{transit2}, \text{move}), (\text{parking2}, \text{deliver}), (\text{transit3}, \text{move})\}\end{aligned}$$ □

Acting with nondeterministic models can be realized by a procedure that, given the current state, applies the action returned by the policy. It consists of observing the current state s, performing the corresponding action $\pi(s)$, and repeating these two steps until the state is no longer in the domain of π. See Algorithm 11.1, which is is similar to Algorithm 8.1, but it simply acts on the states policy and independently of any goal.

In spite of the fact that nondeterministic models can model the world more accurately than deterministic ones (and to plan for recovery mechanisms at design time), they are not necessarily perfect models of the world. Indeed, no model is perfect and the world is seldom completely predicable. As a consequence, if an actor applies the actions of a policy, there is no guarantee that they will all be applicable, nor that they will produce the states predicted by the policy. All the problems mentioned in

[2] Notice that deliver action has two possible effects in one instance and three in another. This is allowed because the degree of nondeterminism can depend on the state in which an action is performed.

[3] For the rest of this chapter, we will use the same names for the states as shown in the figure.

```
Acting with Policy($\pi$)
    $s \leftarrow$ observe the current state
    while $s \in Domain(\pi)$ do
        perform action $\pi(s)$
        $s \leftarrow$ observe the current state
```

Algorithm 11.1. Acting with policies

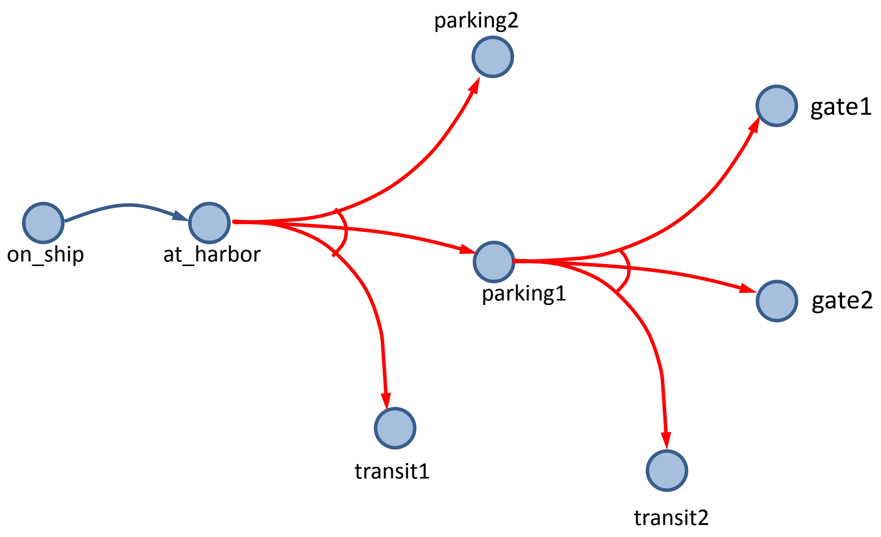


Figure 11.2. Reachability graph for policy π_1.

Chapter 2 can still occur, like execution failures, unexpected events, incorrect and partial information, etc. Similarly to the case of deterministic models, actors need ways to change the policy and to apply different actions when problems are detected. Similarly to what we have done in Chapter 2 for deterministic models, when we encounter a problem, we can generate a new policy (e.g., by replanning).

11.1.2 Unsafe, Cyclic Safe, and Acyclic Safe Policies

From Section 8.1, we recall the notions of the *transitive closure* of $\gamma(s, \pi(s))$ and the *reachability graph* $Graph(s, \pi)$ that connects the reachable states from state s through a policy π. We use a slightly different notion of the *leaves* of a policy π from state s, because we need to guarantee that the leaf node can be reached from s. We let $leaves(s, \pi) = \{s' \mid s' \in \widehat{\gamma}(s, \pi) \text{ and } s' \notin Domain(\pi)\}$. Notice that $leaves(s, \pi)$ can be empty, that is, there may be no leaves. This is the case of policies that cycle on the same set of states. If π is empty, then $leaves(s, \pi) = \{s\}$.

Example 11.3. Let π_1, π_2, and π_3 be as in Example 11.2. Their leaves from the state

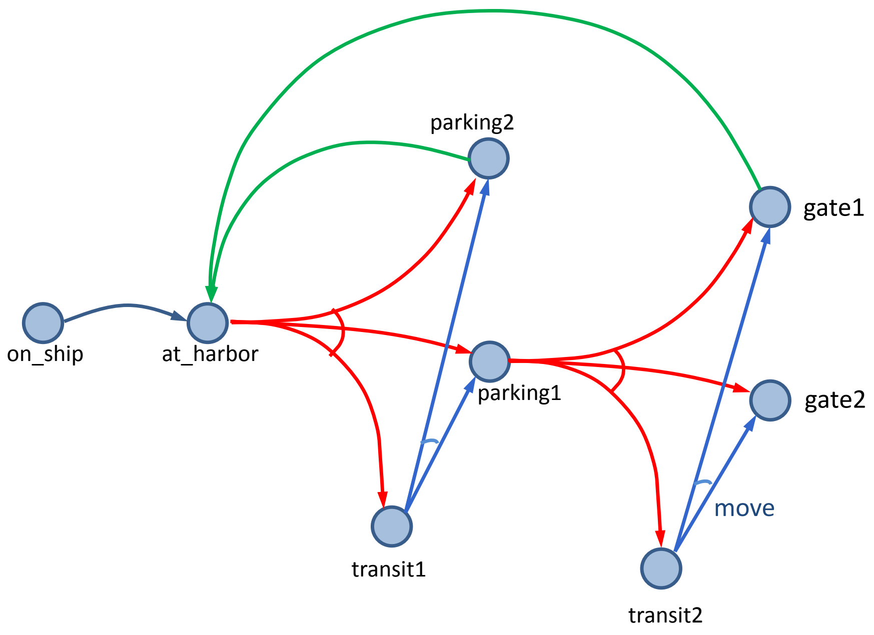


Figure 11.3. Reachability graph for policy π_2.

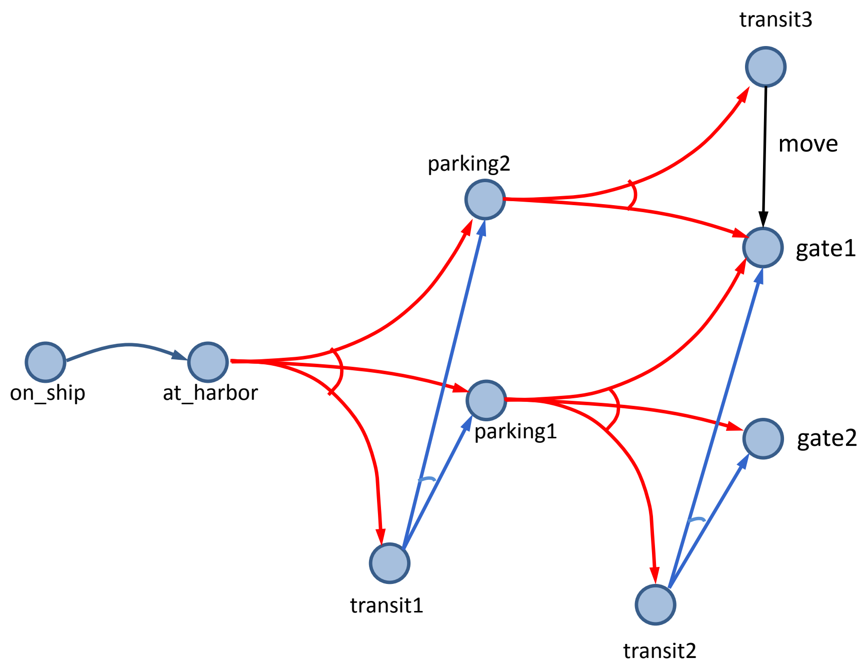


Figure 11.4. Reachability graph for policy π_3.

on_ship are:[4]

$$\begin{aligned}
\mathit{leaves}(\text{on\_ship}, \pi_1) &= \{\text{parking2}, \text{transit1}, \text{gate1}, \text{gate2}, \text{transit2}\}\\
\mathit{leaves}(\text{on\_ship}, \pi_2) &= \{\text{gate2}\}\\
\mathit{leaves}(\text{on\_ship}, \pi_3) &= \{\text{gate1}, \text{gate2}\}
\end{aligned}$$

[4]In this case, the value on_ship of the state variable pos(item) identifies a single state.

Figures 11.2, 11.3, and 11.4 show the reachability graphs of π_1, π_2, and π_3 from the state on_ship. Notice that in each case, all states are reachable from on_ship. □

Given these preliminary definitions, we can now introduce formally the notion of a problem and solution policy in a nondeterministic model. Solutions for nondeterministic models can be defined with respect to an initial state s_0[5] and a set of goal states S_g:

- A policy π is a *solution policy* if and only if $leaves(s_0, \pi) \cap S_g \neq \emptyset$
- A policy π is a *safe policy* if and only if $\forall s \in \widehat{\gamma}(s_0, \pi)(leaves(s, \pi) \cap S_g \neq \emptyset)$
- Let π be a solution policy. Then π is an *unsafe policy* if it is not safe.
- Let π be a solution policy for Σ. Then π is a *cyclic safe policy* if and only if $leaves(s_0, \pi) \subseteq S_g \wedge (\forall s \in \widehat{\gamma}(s_0, \pi))(leaves(s, \pi) \cap S_g \neq \emptyset) \wedge Graph(s_0, \pi)$ is cyclic.
- Let π be a solution policy for Σ. Then π is an *acyclic safe policy* if and only if $leaves(s_0, \pi) \subseteq S_g \wedge Graph(s_0, \pi)$ is acyclic.

Solution policies may lead to a goal. They can achieve the goal in different ways, with different levels of guarantee, and with different strengths. The requirement we impose on a policy to be a solution is that at least one state of its leaves is a goal state. *Safe policies* are policies in which the goal is reachable from the initial state.[6] *Unsafe policies* either have a leaf that is not in the set of goal states or there exists a reachable state from which it is not possible to reach a leaf state. Intuitively, unsafe policies may achieve the goal but are not guaranteed to do so. If an agent tries to perform the actions dictated by the policy, the agent may end up at a nongoal state or end up in a "bad cycle" where it is not possible to go out and reach the goal.

Acyclic Safe Policies are safe policies that are guaranteed to terminate and to achieve the goal despite nondeterminism. They are guaranteed to reach the goal in a bounded number of steps, and the bound is the length of the longest path in $Graph(s_0, \pi)$. Notice that *acyclic* and *cyclic* safe solutions are very different! To guarantee reaching the goal in a bounded number of steps, i.e., without allowing for the possibility to get stuck in a loop, we need acyclic solutions. Cyclic policies may loop forever without reaching the goal!

Notice that, while the notion of safe and unsafe solutions are similar with those in Chapter 8, the definition of safe policies as policies that have probability one to reach the goal (see Definition 8.6) does not allow us to guarantee reaching the goal in a bounded number of steps without excluding the case in which we get stuck in a loop and therefore never reaching the goal.

[5]Notice that we have a single initial state s_0 rather than a set of initial states $S_0 \subseteq S$. A set of initial states represents partially specified initial conditions, or in other words uncertainty about the initial state. However, restricting to a single initial state is not a limitation, because a domain with a set of initial states S_0 is equivalent to a domain where we have a single initial state $s_0 \notin S$ and an additional action $a_o \notin A$ such that $\gamma(s_0, a_0) = S_0$.

[6]Notice that, in general, they are not policies in which the goal is reachable from any state of the domain of the policy ($Domain(\pi)$), because we may have a state in $Domain(\pi)$ that is not the initial state and from which we do not reach the goal.

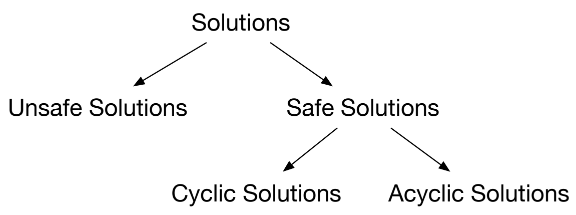


Figure 11.5. Different kinds of solutions: class diagram.

Figure 11.5 depicts in a class diagram the different forms of solutions. In principle, *unsafe Policies* are not of interest, because they do not guarantee to achieve the goal. However, as we will see in Section 12.2, planning for (possibly unsafe) solutions can be used by planning algorithms to guide the search for Safe Solutions. In general, we are interested in safe (cyclic and acyclic) solutions, because they provide (with different strengths) some assurance to achieve the goal despite nondeterminism. *Acyclic Safe Policies* are the best solutions because they can really ensure that we get to the goal. *Cyclic Safe Policies* provide a weaker degree of assurance to achieve the goal: assuming that sooner or later execution will get out of possibly infinite loops, they are guaranteed to achieve the goal. They guarantee that there is always a possibility to terminate the loop. However, for some applications, this may be not enough. For example, if an electrical device has an unreliable power button, then you might need to press the button repeatedly to turn the device on. But if the button is completely broken, then pressing it repeatedly will never turn the device on. In safety critical applications, safe cyclic policies may not be acceptable.

Example 11.4. Consider the three policies π_1, π_2, and π_3 in Example 11.2. Consider the the model Σ described in Example 11.1, initial state s_0 where on_ship, and goal states $S_g = \{\text{gate1}, \text{gate2}\}$.

All three policies are solutions for the problem; indeed there exists at least one leaf state that is in the set of goal states.

Policy π_1 is an unsafe solution because there are leaves that do not belong to S_g from which it is impossible to reach the goal: such leaves are the states where parking2, or transit1, or transit2.

Policies π_2 and π_3 are safe solutions. Policy π_2 is a safe cyclic solution because from each state in its graph it is possible to reach a state in the goal (gate2). Policy π_3 is a safe acyclic solution because it is guaranteed to reach one of the two gates, gate1 or gate2, without the danger of getting trapped in cycles.

Notice that for the same domain, the same initial state, but with goal $S_g = \{\text{gate2}\}$, a safe acyclic solution does not exist, and the safest solution we can find is the safe cyclic solution π_2. □

A remark is in order. We require that solution policies have some leaf states. In this way, we do not consider policies that lead to the goal and then loop inside the set of goal states. One may argue that such policies might be considered as solutions. However, notice that for any solution of this kind, there exists a solution according to our definition. It is indeed enough to eliminate the states in the policy that lead to the loop inside the set of goal states.

In the following, we specify the relations among different kinds of policies.

$$\begin{aligned}&\textit{unsafe policies} \cup \textit{safe policies} = \textit{policies}\\&\textit{cyclic safe policies} \cup \textit{acyclic safe policies} = \textit{safe policies}\\&\textit{unsafe policies} \cap \textit{safe policies} = \varnothing\\&\textit{cyclic safe policies} \cap \textit{acyclic safe solutions} = \varnothing\end{aligned}$$

11.2 Automata

In this section, we introduce the notion of finite state nondeterministic automata, also called Finite State Machines. We focus on input/output automata, i.e., on automata that, beyond states and transitions, represent inputs to and outputs from automata. The underlying intuition is that input/output automata act and evolve by receiving inputs from the environment, sending output to the environment, and by internal transitions. Acting is performed by making the automaton evolve: interacting with the environment by receiving inputs and sending outputs, and through internal transitions. Input/output automata have different ways to represent and deal with nondeterministic models:

- An input/output automaton can specify the different inputs that it may receive from the environment in a given state. Each of them may lead to a different state. However, the automaton cannot control which input it will actually receive. For instance, different inputs can be different data from sensors, like different images sensed by a camera of a robot, or different choices made by a user, or different moves by an opponent.
- Internal transitions may be nondeterministic, i.e., lead to one among possibly many states. This is similar to what happens in nondeterministic state transition systems. For instance, an automaton can represent with a nondeterministic internal action the nondeterministic success or failure of an action by a robot, or the internal check for the availability of a room of the hotel, or the availability of a seat in a flight.

Outputs are controlled by an input/output automaton, i.e., the automaton can decide to send different outputs in different states. They correspond to commands that an acting automaton can send to an execution platform. According to the input output automata view, acting is a continuous stream of interactions between an actor and the environment: the actor acts by sending outputs to the environment, receives inputs from the environment and reacts to such nondeterministic uncontrollable inputs by evolving through internal transitions and by sending further outputs, and so on and so forth.

Input/Output automata allow for specifying distributed systems: An actor can be described as a set of input output automata interacting among themselves and with the environment. Therefore inputs and outputs, beyond being received and sent from/to the environment, can be received from and sent to other automata.

We define the notion of control automaton, i.e., an automaton that acts by controlling the behavior of possibly many different automata that interact among themselves and with the environment.

11.2.1 Input Output Automata

We introduce the intuitions underlying input/output automata through some simple examples. We start with Example 11.5, an example of harbor management. A system dealing with unloading containers from a ship, parking them and delivering them to gates. Example 11.5 allows us to show the similarities and differences with the state transition system representation in Example 11.1, Section 11.1,

Example 11.5. In Figure 11.6, a DWR robot performs the operations for unloading, parking and delivering containers. It interacts with an harbor management system, which in this case, from the point of view of the AVS, acts as the environment. The AVS unloads a container (a deterministic internal action), then asks the management system to have permission to park the container in one of the two parking lots through the output transition parking. The AVS waits for three possible inputs from the environment indicating that either the container can be parked in one of the two parking lots (parking1 or parking2) or it should wait in a transit area (transit1). Here is how the nondeterminism of the state transition system in Example 11.1, Section 11.1 is represented, i.e. waiting for one of different possible inputs from the environment. From the parking lots the AVS asks for a permission to deliver the container to a gate, and again nondeterminism is represented with inputs received by the environment (transit2, transit3 gate1, gate2) □

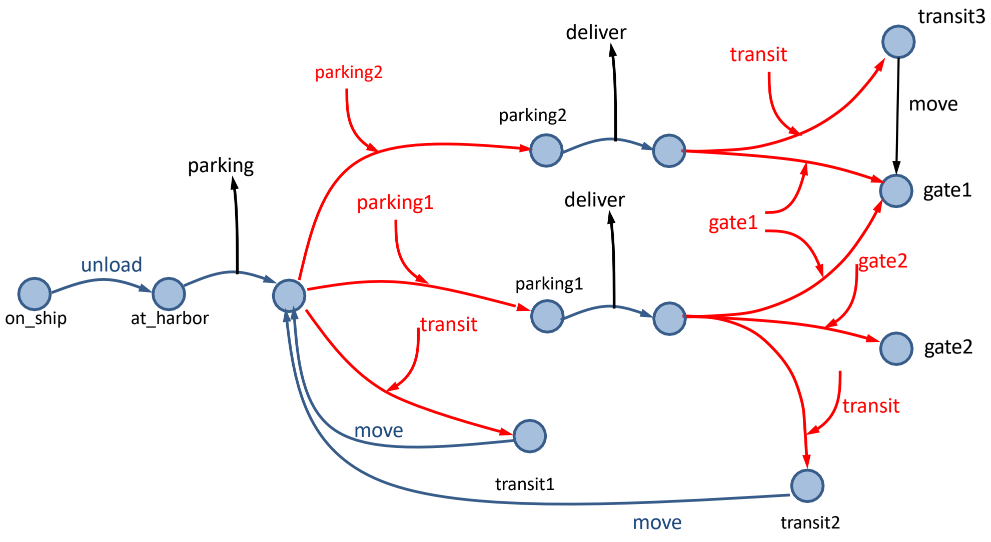


Figure 11.6. Input/output automaton for harbor management.

Let us now illustrate distributed input/output automata. Suppose a robot navigates in an environment with different kinds of doors that can be opened differently, e.g., by pulling, by pushing, or sliding. To go through a door, the robot needs recognize its type. Rather than equipping the robot with this recognition capability, assume that doors are able to informe about their type.[7] In some way, we "distribute the intelligence" in the environment. The task for the robot becomes much simpler, and it can be described in the following example.

Example 11.6. In Figure 11.7, the robot gets the door's type, for example, by sending a request to the door, which replies with information about the way the door can be opened. Notice that we have three different kinds of transitions in the nondeterministic model triggered by the request sent by the robot to the door, the output message (door_type), and the answer received from the door, the input messages (pulling, pushing, and sliding). This is a way to define a nondeterministic model through outputs sent by the automaton and inputs received by the automaton. □

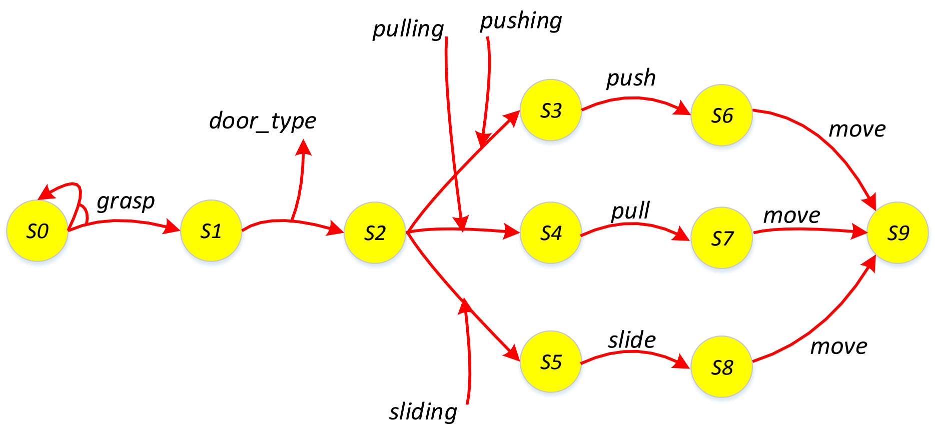


Figure 11.7. Input/output automaton for opening a door

As introduced informally in Example 11.5 and Example 11.6, the idea is to specify nondeterministic models as *input/output automata*, the main feature of which is to model components that interact with each other through inputs and outputs. Input/output automata allow for modeling distributed systems where each automaton is a component that interacts with other components through inputs and outputs. They make it possible to simplify the design process by abstracting away the details of their internal representation.

Formally, input/output automata are very similar to nondeterministic models described in Section 11.1, with the following differences. Input/output automata can evolve to new states by receiving *inputs* from other automata and sending *outputs* to other automata. Moreover, they can evolve with *internal transitions* without sending outputs and receiving inputs. Internal transitions can be nondeterministic.[8]

[7] E.g., with an RFID stick, cheaper than automating doors.

[8] In automata theory, the symbol τ is used to denote internal transitions, which are called τ-transitions. The reason is that usually internal transitions are not visible to other automata. In our representation, we distinguish internal transitions that are triggered by different actions.

Definition 11.7. (Input/Output Automaton) An input/output automaton is the tuple $Aut = (S, S^0, I, O, A, \gamma)$, where

- S is a finite set of states;
- $S^0 \subseteq S$ is the set of possible initial states in which the automaton can start;
- I is the set of inputs, O is the set of outputs, and A is a set of actions, with I, O, and A disjoint sets;
- $\gamma : S \times (I \cup O \cup A) \rightarrow 2^S$ is the nondeterministic state transition function. □

The distinction between inputs and outputs is a main characteristics of input/output automata. The intended meaning is that outputs are under the full control of the automaton, that is, the automaton can decide when and which output to send. In contrast, inputs are not under its control. If and when they are received, which input is received (among many different ones) from other automata cannot be determined by the automaton receiving inputs. An automaton can wait for the reception of an input, but whether it will receive it, and when it will receive it, is not under its control.

While acting, the different possible evolutions of an input/output automaton can be represented by its set of possible *runs*.

Definition 11.8. (Run of input/output automaton) A *run* of an input/output automaton $Aut = (S, S^0, I, O, A, \gamma)$ is a sequence $s_0, a_0, s_1, a_1, \ldots$ such that $s_0 \in S^0$, $a_i \in I \cup O \cup A$, and $s_{i+1} \in \gamma(s_i, a_i)$. □

A run may be either finite or infinite.

Control Automaton

An input/output automaton can behave in different ways depending on the inputs it receives. For instance, the automaton in Figure 11.7 opens the door either by pushing, pulling, or sliding the door, on the basis of the input it receives. In order to control (a set of)em, e.g., to reach some desired states. to restrict their evolutions in a desired way, we define a *a controller* or *control automaton*, that is, an automaton that interacts with them by reading their outputs and sending them inputs. A control automaton Aut_c for an input/output automaton Aut is an input/output automaton whose inputs are the outputs of Aut and whose outputs are the inputs of Aut. Formally: Let $Aut = (S, S^0, I, O, A, \gamma)$ be an input/output automaton. A control automaton for Aut is an input/output automaton $Aut_c = (S_c, S_c^0, O, I, A_c, \gamma_c)$.

Example 11.9. Figure 11.8 shows a control automaton for the automaton in Figure 11.7. Notice that the inputs and outputs of the automaton in Figure 11.8 are the outputs and inputs of the automaton in Figure 11.7, respectively. The control automaton receives a request about the door type (input door_type), determines the door type with the action sense(door), and sends the proper input to the controlled automaton. The information acquisition about the door type can be done in different ways. The nondeterministic action sense(door) can activate a module of an "intelligent" door that replies to requests by the robot, or sense(door) can be a request to a centralized database that may have apriori the information about the door, or sense(door) might

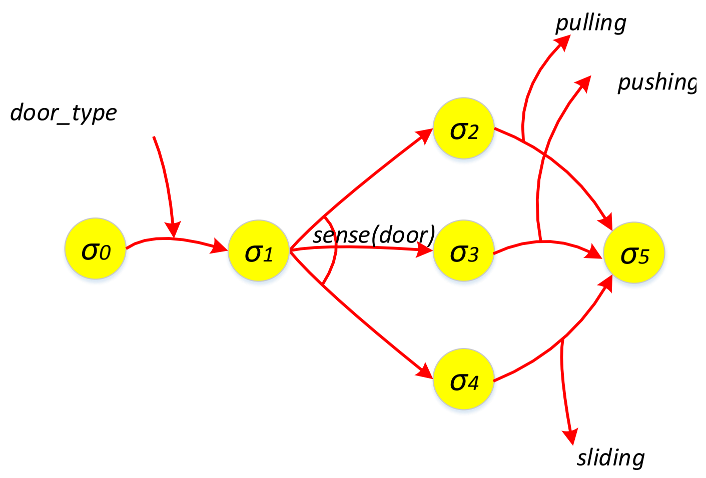


Figure 11.8. Input/Output (I/O) automaton to control the robot I/O automaton.

activate a different module that has some perception capabilities to detect the kind of door. □

Controlled Automaton

A control automaton can control more than one input/output automata. This is the way in which we can control a distributed system. We will address this issue in Section 11.2.2. For simplicity, let us consider a control automaton controlling a single automaton. It is interesting to understand the behavior of an automaton when controlled by a control automaton, that is, the behavior of the *controlled system*. The controlled system should be able to satisfy some desired property. Indeed, a control automaton is designed with a goal in mind. For instance, the automaton in Example 11.9 has been defined with the requirement in mind to open the door in the right way. In the automaton in Figure 11.7, it means to end up in state s_9. Notice that such automaton just represents the nominal case. If it receives a wrong input, for example, to pull a door that should be pushed, then the move action will fail. Consider the following example, which helps us to give an intuition of the desired behavior of a controlled system

Example 11.10. In Figure 11.9, the input/output automaton of the robot checks whether it is close enough to the door (action sensedistance) and sends outputs accordingly. If it is far, it can receive the input either to wait or to move (state s3). Let us suppose the goal is to make the automaton reach state s5. It is clear that a control automaton that receives the input far from the automaton in Figure 11.9 and sends output wait does not satisfies the goal, while the one that sends the sequence of outputs move and then grasp does.

Notice that we may have a control automaton that never makes the controlled automaton reach state s5, or that do it only in one of the two cases in which the robot

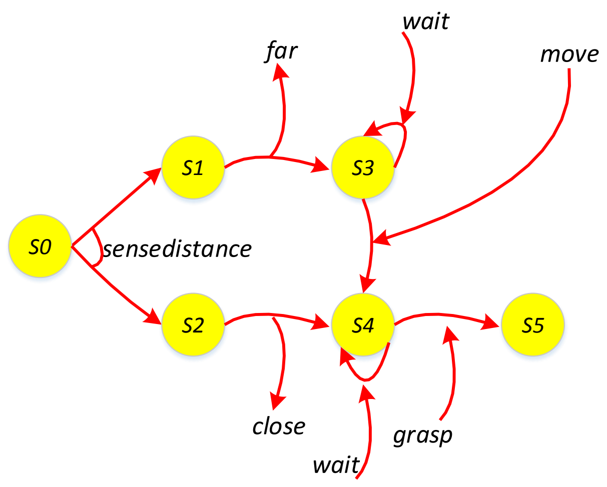


Figure 11.9. Input/output automaton for approaching a door.

is close or far, or that do it in both cases. All of this resembles the idea of unsafe and safe (cyclic and acyclic) policies introduced in Section 11.1.2. □

As shown in the previous example, we would like that a control automaton Aut_c that interacts with an input/output automaton Aut satisfies some goal g. In this section, we restrict to reachability goals.[9] Let us then define now the automaton describing the behaviors of Aut when controlled by a control automaton Aut_c , that is, the *controlled system* $Aut_c \rhd Aut$.

Definition 11.11. (Controlled System) Let $Aut = (S, S^0, I, O, A, \gamma)$ be an input/output automaton. Let $Aut_c = (S_c, S_c^0, , O, I, A_c, \gamma_c)$ be a control automaton for Aut. Let $s, s' \in S$, $s_c, s'_c \in S_c$, $a \in A$, and $a_c \in A_c$. The controlled system $Aut_c \rhd Aut$, describing the behavior of Aut when controlled by Aut_c, is defined as: $Aut_c \rhd Aut = (S_c \times S, S_c^0 \times S^0, I, O, A_\rhd, \gamma_\rhd)$, where:

- $\langle s'_c, s\rangle \in \gamma_\rhd(\langle s_c, s\rangle, a_c)$ if $s'_c \in \gamma_c(s_c, a_c)$,
- $\langle s_c, s'\rangle \in \gamma_\rhd(\langle s_c, s\rangle, a)$ if $s' \in \gamma(s, a)$,
- for any $i \in I$ from Aut_c to Aut,
  $\langle s'_c, s'\rangle \in \gamma_\rhd(\langle s_c, s\rangle, i)$ if $s'_c \in \gamma_c(s_c, i)$ and $s' \in \gamma(s, i)$,
- for any $o \in O$ from Aut to Aut_c,
  $\langle s'_c, s'\rangle \in \gamma_\rhd(\langle s_c, s\rangle, o)$ if $s'_c \in \gamma_c(s_c, o)$ and $s' \in \gamma(s, o)$. □

The set of states of the controlled system are obtained by the Cartesian product of the states of Aut and those of Aut_c. In Definition 11.11, the first two items specify that the states of the controlled system evolve according to the internal evolutions due to the execution of both actions of Aut_c (first item) and of actions of Aut (second item). The third and fourth items regard the evolutions that depend on inputs and outputs.

[9] As usual, we consider goals as partial assignments to state variables.

In this case, the state of the controlled system $\langle s_c, s \rangle$ evolves by taking into account the evolutions of both Aut and Aut_c.

A remark is in order. We need to rule out controllers that can get trapped in deadlocks. We need to rule out the case in which an automaton sends outputs that the other automaton is not able to receive. If an automaton sends an output, then the other automaton must be able to consume it, either immediately or after executing internal commands that lead to a state where the input is consumed. In other words, an automaton Aut in a state s must be able to receive as one of its inputs $i \in I$ the output $o' \in O'$ of another automaton Aut', or for all the possible executions of actions $a \in A$ of automaton Aut, there exists a successor of s where o' can be received as an input i.

Given this notion, we define intuitively the notion of a *deadlock-free controller* for a controlled input/output automaton. It is a control automaton such that all of its outputs can be received by the controlled automaton, and vice versa, all the outputs of the controlled automaton can be received by the controller.

Solution Control Automata

We aim at designing a control automaton Aut_c such that the controlled system $Aut_c \triangleright Aut$ satisfies a goal g, that is, we have to design an Aut_c that interacts with Aut by making Aut reach some desired state. In other words, a control automaton Aut_c is a solution for a goal g if every run of the controlled system $Aut_c \triangleright Aut$ ends up in a state where g holds.

Definition 11.12. (Satisfiability). Let g be a partial state variable assignment $x_i = v_i, \dots, x_k = v_k$, for each $x_i, \dots, x_k \in X$, and each $v_i \in Range(x_i), \dots, v_k \in Range(x_k)$. Let Aut be an input/output automaton. Aut satisfies g, denoted with $Aut \models g$, if

- there exists no infinite run[10] of Aut, and
- every final state s of Aut satisfies g. □

We can now define when a control automaton is a solution for an input/output automaton with respect to a goal, that is, when it controls the automaton satisfying our desired requirement.

Definition 11.13. (Solution Control Automaton). A control automaton Aut_c is a solution for the goal g and an input/output automaton Aut, if the controlled system $Aut_c \triangleright Aut \models g$ and Aut_c is a deadlock-free controller for Aut. □

11.2.2 Distributed Input Output Automata

In the previous section, we did not exploit the real advantage of the distributed and asynchronous nature of input/output automata. Indeed, Example 11.6 may include two input/output automata, one for the robot and one for the door. The two automata interact by sending/receiving inputs/outputs. The main characteristic of a model based

[10]See Definition 11.8 for the definition of run of an automaton

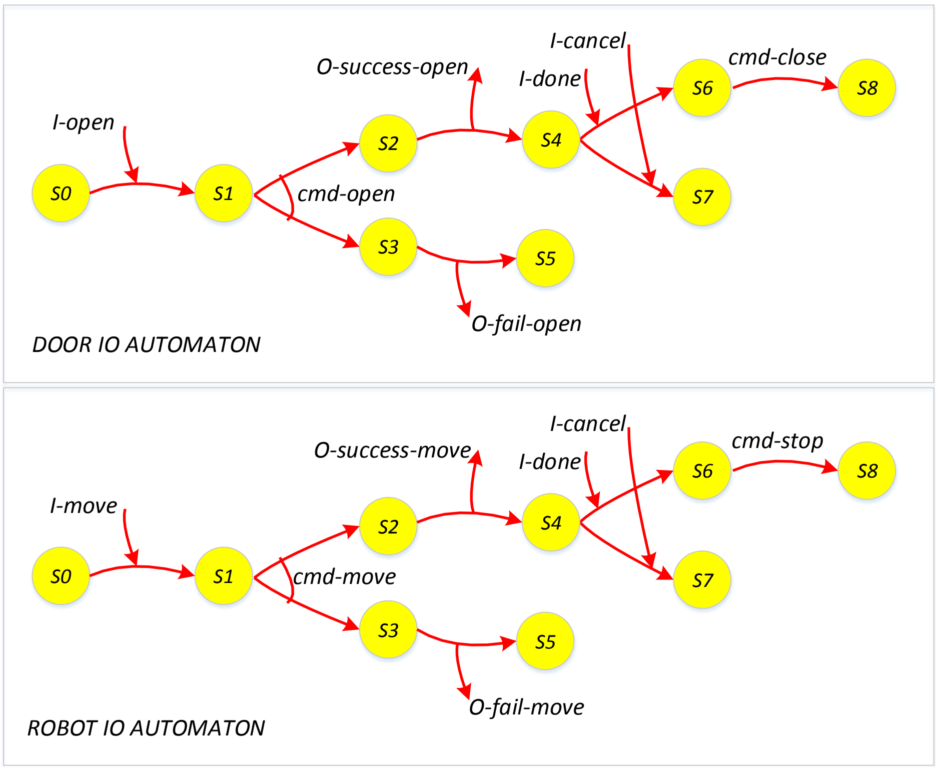


Figure 11.10. Robot and door interacting input/output automata.

on input/output automata is that a complex model can be obtained as the "composition" of much simpler components, thus providing the following advantages:

- the ability to simplify the design process, starting from simple components whose composition defines a model of a complex system;
- the ability to model distributed domains naturally, that is, domains where we have different components with their own behaviors;
- the ability to model naturally dynamic environments when different components join or leave the environment;
- the composition of different components can be localized, that is, each component can get composed only with the components that it needs to interact with, thus simplifying significantly the design task, and
- for each component, we can specify how other components need to interact with the component itself, abstracting away the details of their internal operations.

Example 11.14. We continue with our example of opening a door, but we reduce the tasks that can be performed by the robot while we enrich the autonomy capabilities of the door. Consider indeed two active devices that interact with the environment, a navigation robot able to move but without any manipulation capabilities, and an active door, which is able to open and close itself on request.

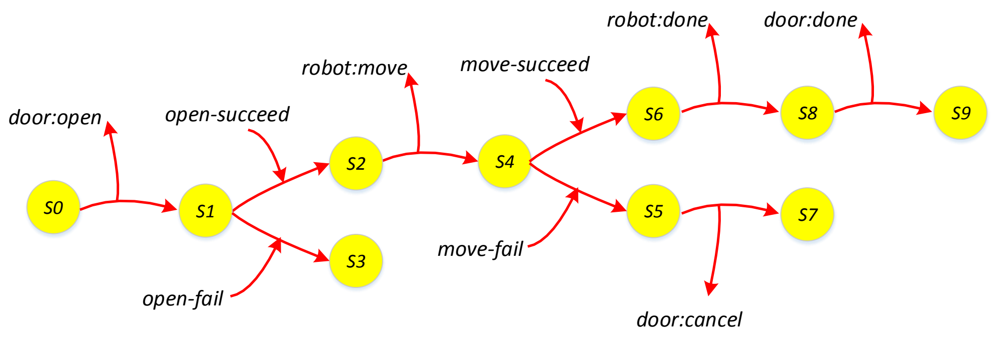


Figure 11.11. A controller for the robot and door input/output automata.

In Figure 11.10, the door and the robot are modeled as two input/output automata. The door receives a request to open (the input I-open). It then activates its engines to open (the actions cmd-open). The action may either succeed or fail, and the door sends outputs accordingly (O-succes-open or O-fail-open). If the action succeeds, then the door waits for two possible inputs, one indicating that the door can be closed (because, e.g., the robot passed successfully), that is, the input I-done, or that there is a problem and the door should stop with failure, that is, I-cancel. The (I-move), then it moves (cmd-move). If the operation succeeds, then it waits for an input stating either that everything is fine (I-done) and it can stop (cmd-stop) or that a failure from the environment occurred (I-cancel). □

Notice that with this model, the robot and any other actor in the environment do not even need to know whether the door is a sliding door or a door that can be opened by pulling/pushing, because this is hidden in the action cmd-open of the door input/output automaton. This abstraction mechanism is one of the advantages of a model based on multiple input/output automata.

Given a model with two or more input/output automata, we generalize the idea presented for controlling a single automaton, that is, a controller that interacts with the different input/output automata and satisfies some goal. Consider the following example.

Example 11.15. Figure 11.11 shows an input/output automaton representing a controller that makes the robot and the door interact in a proper way. It requests that the door open, and if the request succeeds, it then requests the robot to move. If the moving operation also succeeds, it then asks the door and the robot to finish the job, that is, the robot should stop and the door should close. □

In the rest of this section, we formalize the problem of defining a controller that interacts with a set of input/output automata $Aut_1, \ldots, Aut_n$ and satisfies some desired goal. We have:

- A finite set of automata $Aut_1, \ldots, Aut_n$. This set can be dynamic and can be determined at run-time.
- A requirement g that is defined as a partial state variable assignment.

Informally, we want to design a controller Aut_c that interacts with $Aut_1, \ldots, Aut_n$ in such a way to make the automata $Aut_1, \ldots, Aut_n$ to reach some states where the requirement g is satisfied. We introduce first the product of the automata $Aut_1, \ldots, Aut_n$:

$$Aut_{\|} = Aut_1 \| \ldots \| Aut_n$$

Such product is a representation of all the possible evolutions of automata $Aut_1, \ldots, Aut_n$, without any control by Aut_c.

We formally define the product of two automata Aut_1 and Aut_2, which models the fact that the two automata may evolve independently. In the following definition, we assume that the two automata do not send messages to each other, that is, the inputs of Aut_1 cannot be outputs of Aut_2 and vice versa. This is a reasonable assumption in our case, where we suppose that each available automaton $Aut_1, \ldots, Aut_n$ interacts only with the controller Aut_c. The assumption can, however, be dropped by modifying in a suitable way the definition of *product*.

Definition 11.16. (Product of Input/Output Automata) Let $Aut_1 = (S_1, S_1^0, I_1, O_1, A_1, \gamma_1)$ and $Aut_2 = (S_2, S_2^0, I_2, O_2, A_2, \gamma_2)$ be two automata with $(I_1 \cup O_1 \cup A_1) \cap (I_2 \cup O_2 \cup A_2) = \emptyset$. The product of Aut_1 and Aut_2 is $Aut_1 \| Aut_2 = (S, S_0, I_1 \cup I_2, O_1 \cup O_2, A_1 \cup A_2, \gamma)$, where:

- $S = S_1 \times S_2$,
- $S_0 = S_1^0 \times S_2^0$,
- $\langle s_1', s_2 \rangle \in \gamma(\langle s_1, s_2 \rangle, a)$ if $s_1' \in \gamma_1(s_1, a)$, and
- $\langle s_1, s_2' \rangle \in \gamma(\langle s_1, s_2 \rangle, a)$ if $s_2' \in \gamma_2(s_2, a)$ □

The automaton $Aut_{\|} = Aut_1 \| \ldots \| Aut_n$ represents all the possible ways in which automata $Aut_1, \ldots, Aut_n$ can evolve without any control. We can therefore define the automaton describing the behaviors of $Aut_{\|}$ when controlled by a controller Aut_c that interacts with $Aut_1, \ldots, Aut_n$, that is, the controlled system $Aut_c \triangleright Aut_{\|}$, simply by recasting the definition of controlled system (see Definition 11.11) by replacing the single automaton Aut with $Aut_{\|}$. We can therefore apply all the considerations, definitions, and algorithms that we have discussed for the case of a single automaton.

In Section 12.4 we will show how to synthesize a controller by planning with nondeterministic models.

11.3 Behavior Trees

Behavior Trees (BTs) provide a graphical representation, equivalent in expressiveness to finite state automata. They do not represent states explicitly; they focus instead on state changes or “behaviors”. They allow for an intuitive and modular specification of reactive control mechanisms. Acting is performed by repeatedly visiting a BT from the root until reaching a leaf node that triggers test conditions or primitive actions and returns its execution status (running, failure, or success). Acting with BTs can deal with exogenous events and some form of uncertainty about the environment.

11.3.1 Behavior Tree Representation

A Behavior Tree is a directed rooted tree. Each node v can be a leaf or a non-leaf or interior node. A leaf node can be of type Cond or Act: $type(v) \in \{\text{Cond}, \text{Act}\}$. A leaf node of type Cond corresponds to a condition that can be tested in the observed current state of the world. A leaf node type Act corresponds to a primitive action. A non-leaf/interior node v has $type(v) \in \{\text{And}, \text{Or}\}$[11] and has a totally ordered list of children, $Children\ (v) = \langle v_1, \ldots, v_k \rangle$.

For each node v there is a function exec-status (v) that returns a value in $\{$Running, Success, Failure$\}$. How exec-status(v) is computed depends on $type(v)$:

- exec-status(v) for leaf nodes is computed as follows:
    1. If $type(v)$ = Cond, then a call to exec-status(v) returns Success if the condition is true, or Failure if the condition is false. If $type(v)$ = Cond, then exec-status(v) is never equal to Running, meaning that conditions are immediately computed.
    2. If $type(v)$ = Act, then the first call to exec-status(v) triggers the action and returns Running. Each subsequent call returns Running if the action is still running, or Success or Failure if the action has finished.
- exec-status(v) for non-leaf/interior nodes is computed as a control function over v's children:
    - If $type(v)$ = And, then exec-status(v) returns Success if $\forall v_i \in Children(v)$, exec-status$(v_i)$ = Success. If $\exists v_i$ such that exec-status(v_i) is Failure or Running, the computation stops at the first such i; it returns exec-status(v_i) without testing the remaining children.
    - If $type(v)$ = Or, then exec-status(v) returns Failure if $\forall v_i \in Children(v)$, exec-status$(v_i)$ = Failure. If $\exists v_i$ such that exec-status(v_i) is Success or Running, the computation stops at the first such i and returns exec-status(v_i), without evaluating the rest of v's children[12]

Example 11.17. Consider a Versatile Service Robot (VSR) domain where a robot has to serve a simple jobshop by bringing parts from an input stock to a machining station. Let us use a BT to model the activity in which the robot takes a part in the input stock (stock) and brings it to the machining station (mach). We assume that the the robot can perform the following primitive actions that will be modeled as leaf nodes of type Act in the BT:

- goto(r, l): robot r goes to position l. There are two possible positions: l = stock, where a part may be picked up, and l = mach, where the part can be fed into the machining station.
- take(r, p): robot r takes part p from stock.
- put(r, p): robot r puts part p into mach.

[11] In the BT literature, And and Or nodes are called *sequence* and *fallback* nodes, respectively.

[12] Notice that And and Or nodes do not represent primitive operations, in the sense that one can be defined in terms of the other one

Figure 11.12 gives a BT specification of robot r's activity. If the part p is already in the machine, then the task is done. Otherwise the following And node (node1 in Figure 11.12), through successor Or nodes, checks if r holds p and if r's position is the machining station (nodes 2 and 3), in which case r puts p in mach. If r is not holding p, its arm is free (node 4), and it is at stock, then it takes p. If r is not at stock but stock is reachable (node 6 and 7), then r goes to that position. Similarly, if the condition pos(r)=mach does not hold, r moves to the machining station (node 5). Additional nodes would be needed, for example, to handle the case where the condition hold(r)=free does not hold (see Exercise 11.10).

This BT is reactive to exogenous events; e.g., if an obstacle makes stock or mach unreachable, then r will keep trying until they become reachable (a fancier robot would try to remove the obstacle or ask for help). Similarly, if r takes the part but it slips from r's arm, r will repeat its take action (here too a diagnosis of why the previous grasp failed would be needed). The BT is also reactive to favorable conditions, e.g., if r is already at stock then it does need to move there. □

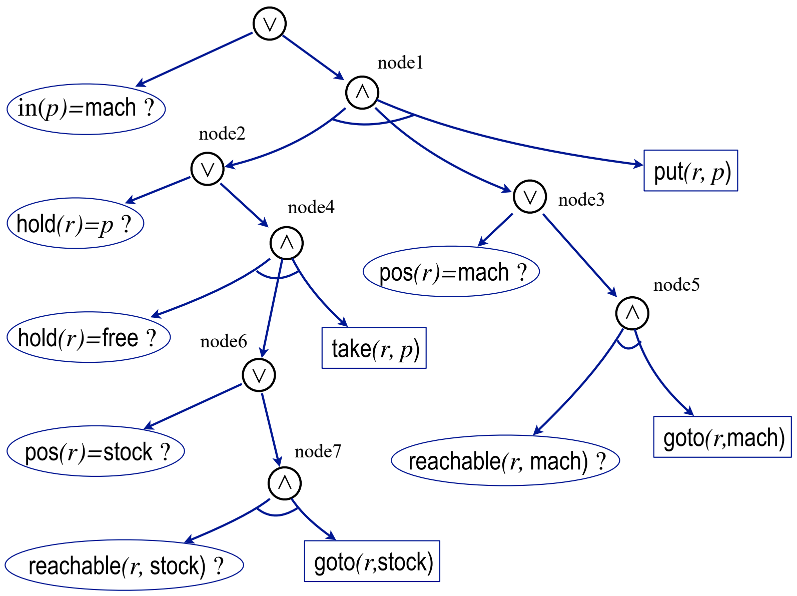


Figure 11.12. A behavior tree specification of the robot activity in Example 11.17. And and Or nodes are respectively labelled ∧ and ∨, condition nodes are in ovals, actions in rectangles.

11.3.2 An Acting Engine for Behavior Trees

Acting with a BT consists of calling BTAE on the root node repeatedly until it returns either Success or Failure. A call is propagated recursively from the root down in the tree until reaching a leaf. A leaf node triggers the corresponding condition or primitive actions and returns its execution status. A Running status is propagated back in the tree up to the root; Failure or Success cases are also propagated back, but may

```
BTAE($v$)
    if $Type(v) \in \{$Cond, Act$\}$ then return exec-status ($v$)
    $\langle v_1, \ldots, v_k \rangle \leftarrow$ Successors($v$)
    for $v_i = v_1$ to $v_k$ do
        $status \leftarrow$ BTAE($v_i$)
        if $status$=Running then return $status$
        if ($Type(v_i)$=And) and ($status$=Failure) then return $status$
        if ($Type(v_i)$=Or) and ($status$=Success) then return $status$
    return $status$
```

Algorithm 11.2. A Behavior Tree Acting Engine, BTAE

trigger possible progress in the sequence of siblings in an interior node, depending on its type. A few remarks are in order:

- The BT execution engine will call v (and hence its children) repeatedly, without retaining any memory of what happened in previous calls. This may work well if one wants the actor to make choices that depend only on the current state (e.g., to keep moving forward until it reaches a goal), but it provides no way for the actor to make choices that depend on past events, unless information about those past events is stored explicitly in the current state.
- Calling v_1 again if it previously returned Success or Failure allows to be reactive to changes of a condition. However, if the BT does not contain all of the relevant information about the action's effects and preconditions when it was previously triggered, it might be unclear whether it should be retriggered.
- A BT may trigger concurrent actions, but offers no explicit means for handling this concurrency. For example, suppose a child v_j of v returns Running because of an action a that has not yet finished. The next time v is called, a child v_i that precedes v_j may trigger another action a' without checking how a' may interfere with the ongoing action a. Note that in this case a' has already been triggered in a previous call.

Additional types of interior nodes with memory have been proposed to handle some of the above issues. However, these memory nodes are not much used since they reduce the reactivity of the BT. Instead, the main fix is a BT programming style that systematically puts a condition node as the first child of each interior node, as illustrated in the following example.

Some of the issues about the repeated calls to the same nodes are handled with the BT specification style illustrated in Example 11.17. This style requires an action node to be preceded by other nodes that test the action's intended effects and preconditions. For a subtree starting in a node v that is intended to achieve a sequence of actions $\pi = \langle a_i, \ldots, a_j \rangle$, with Children($v$)=$\langle v_1, \ldots, v_k \rangle$, there are two cases:

- if v is an Or node, then v_1 is a condition node or a subtree that tests the intended effects of π,
- if v is an And node, then v_1 and possibly its following sibling are nodes or

subtree that test or achieve the preconditions of π.

Acting with a BT relies on a single grounded tree that specifies the entire behavior. This global tree can be composed of several subtrees that specify different subtasks. The composition operation requires condition nodes or subtrees to be carefully set up. Further, BTs are not as flexible as that of the refinement methods in Section 14.2.

In summary, BTs are a simple graphical representation, equivalent to finite state automata. It is possible to automatically map a BT into the equivalent automaton and use related methods for its analysis. The main advantages of BTs are their intuitive programming style and the capability to generate a tree online, interleaving acting and planning. This idea will be developed in Section 12.5.

11.4 Petri Nets

Petri Nets (PNs) are an expressive class of automata and a graphical representation that has been developed for modeling asynchronous concurrent state transition systems. They can model activities with concurrency, precedence and synchronisation constraints and analyze their correctness. A PN is a directed graph with anonymous labels on vertices called *tokens*. The current state of a system modeled by a PN is given by the token distribution in the net. The edges connecting vertices describe what state transitions may take place. Formal developments have led to powerful methods for analyzing the correctness of PN models and proving properties such as liveliness, boundedness and reachability conditions. These theoretical and practical analysis tools ease the development and debugging of models of acting systems where concurrency is important. They provide an advantage in safety critical applications.

In this section we introduce the PN representation, present a PN acting system, illustrate how to model actions and tasks with PNs, then briefly discuss verification and validation issues.

11.4.1 Petri Net Representation

A Petri Net is a directed graph with two classes of vertices, called *places* and *transitions*. It is a bipartite directed graph: there are no edges between vertices of the same class, only from places to transitions, or from transitions to places. A place can hold zero or several *tokens*. A transition is *enabled* if all its parent places hold at least one token. When an enabled transition is *fired*, the tokens are decreased by one in each parent, and increased by one in each successor. In general, no conservation is required in the total number of tokens from parents to successors. The firing of enabled transitions models the dynamics of a discrete-event system.

More formally, let $\mathcal{G} = (Places, Transitions, Edges, \mu)$ be a bipartite directed graph whose vertices are $Places \cup Transitions$ such that:

- $Edges \subseteq (Places \times Transitions) \cup (Transitions \times Places)$, we denote *In(t)* the parent places of a transition t, *Out(t)* are its successors; and
- $\mu : Places \longrightarrow \mathbb{N}$ is a *marking* of $\mathcal{G}$, that is a distribution of tokens in *Places*, $\mu(p) \in \mathbb{N}$ is the current number of tokens in the place p.

Firing transitions. The markings of a Petri Net $\mathcal{G}$ evolve according to the following rules:

(i) a transition t is *enabled* if $\mu(p) \geq 1$ for $\forall p \in In(t)$,

(ii) firing an enabled transition changes μ into μ' such that

- $\forall p \in In(t), \mu'(p) = \mu(p) - 1$,
- $\forall p \in Out(t), \mu'(p) = \mu(p) + 1$, and
- $\forall p \notin In(t) \cup Out(t), \mu'(p) = \mu(p)$

(iii) enabled transitions are fired sequentially, only one at a time.

Note that if $p \in In(t) \cap Out(t)$ (a loop) then $\mu'(p) = \mu(p)$.

At any time, there may be several enabled transitions in a PN that can be fired. Two transitions t and t' that are enabled at the same time by common parents, i.e., $In(t) \cap In(t') \neq \emptyset$, are called *conflicting transitions*. If t and t' are not conflicting, firing t does not disable t', which can be fired next. But if t and t' are conflicting, choosing nondeterministically and firing t may disable t'. With such a mechanism, a PN can represent the dynamics of a nondeterministic state transition system.

Marking graph. A Petri Net (like a behavior tree) does not represent states explicitly but through the token distribution given by μ. The marking μ characterize the current state. μ can be written as a state vector of $|Places|$ integers.[13] The i^{th} component $\mu(p_i)$ models a state variable ranging over the integers.

To each PN corresponds a *marking graph*, i.e., a graph whose nodes are possible markings and edges given by a transition function γ defined as follow:

$$\gamma(\mu, t) = \mu', \text{the marking obtained by firing a transition } t \text{ enabled in } \mu;$$
$$\gamma(\mu, t) = \mathsf{nil} \text{ if } t \text{ is not enabled in } \mu.$$

The state space of a PN is its marking graph. It may be infinite. For example, in a PN with a loop of two edges from p to t and from t to p, and an edge from t to p', firing t when enabled keeps $\mu(p)$ stable but increases $\mu(p')$ indefinitely.

Binary Petri Nets Let us now focus on a class of *binary Petri Nets* where $\mu(p)$ is restricted to $\{0,1\}$. Here a transition of t is enabled if all its parents hold a token. Firing an enabled transition sets all its parents to 0 and all its successors to 1, regardless of their previous values. If p is the parent as well as the successor of t (a loop), p remains at 1 when t is fired. Binary PNs can model *on*/*off* activities or processes with places, and events with transitions. An activity is *on* when it holds a token.

Edges that have several parents or several successors allow modeling different state transition mechanisms, as illustrated in the following example.

Example 11.18. Figure 11.13 shows three binary PNs. Places and transitions are depicted respectively as circles and rectangles; tokens as dots within a place. The figures illustrate three cases of multiple parents or successors.

[13]Firing rules are easily expressed as vector operations, convenient for verification purposes.

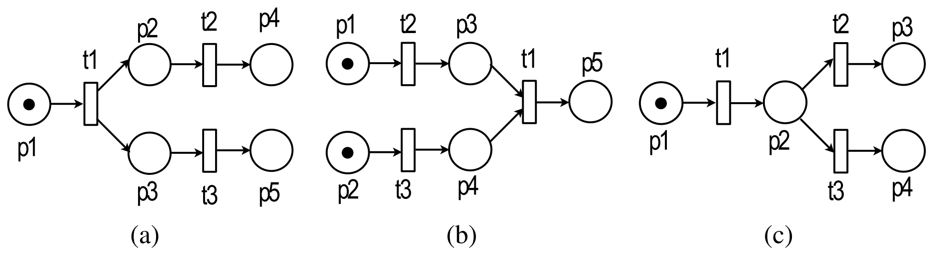


Figure 11.13. Three binary PNs: (a) t1 is a *fork* transition that triggers two concurrent processes; (b) t1 is a *join* transition between two concurrent processes; (c) models a nondeterministic choice: after t1 is fired, both t2 and t3 are enabled, but only one them can be fired.

Figure 11.13(a) models a *fork* transition. t1 is enabled. When fired, it triggers two concurrent processes p2 and p3, which will be followed respectively by p4 and p5.

Figure 11.13(b) models a *join* transition: p1 then p3 runs concurrently with p2 then p4. At that point t1 is enabled. It triggers a single activity in p5.

Figure 11.13(c) models *nondeterminism*: after the firing of t1 both t2 and t3 are enabled but conflicting. A nondeterministic choice has to be made on pursuing with either p3 or p4. Note that in the PNs (a) and (b), both t2 and t3 may be enabled but they are not conflicting; the order in which they are fired is not critical.

The case (not depicted) of a place p with two parent transitions models a disjunction of two events either of which tiggers the activity corresponding to p. □

It is convenient to note a marking μ of a binary PN by the subset of places that hold a token in μ, as illustrated next

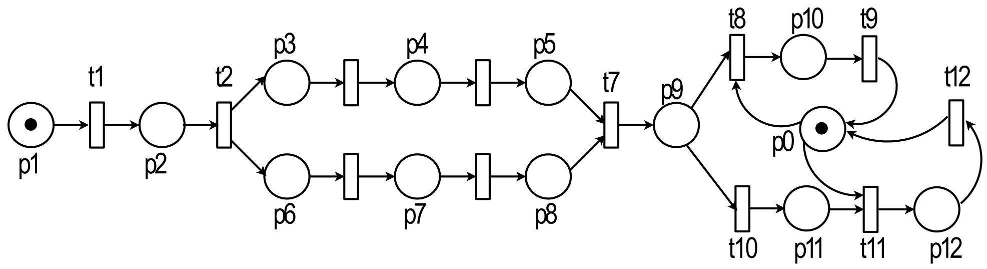


Figure 11.14. A binary Petri Net modeling two simple concurrent processes and a nondeterministic choice.

Example 11.19. Figure 11.14 models a simple packaging domain. A token in p1 means that a new order arrives. This order is preprocessed (p2) followed by two concurrent processes, machining (from p3 to p5), and package preparation (from p6 to p8). When both finishes, p9 holds a token. It corresponds to an inspection activity. Depending on its result, the package is either shipped (p10) or recycled (p11).

Let us analyze how the marking of this PN evolves. The initial marking $\mu_{0,1}$ shown in Figure 11.14 enables only t1, since t8 and t11, successors of p0, have parents without tokens. After firing t1, transition t2 is enabled. It has two successors. t2 is a *fork* allowing the two subnets starting in p3 and p6 to run concurrently. Symmetrically, t7 has two parents. It synchronizes the ending of the two concurrent subnets (machining and package preparation) into a *join*, enabled when both p5 and p8 have a token. Sequential firing resumes with t7, leading to the marking $\mu_{0,9}$ with one token in p9 and the original one in p0.

The two successors of p9 are enabled in $\mu_{0,9}$, but *conflicting*; only one transition can be fired. The nondeterministic choice between t8 and t10 may be conditioned on an external test (e.g., the inspection activity in the packaging domain), or it may rely on a heuristic, or a look-ahead to tell which choice is preferable on the long run for a utility criteria or for reaching some goal marking.

Place p0 has two successors as well as two parents. It might also involve a nondeterministic choice if both p9 and p11were marked, enabling the conflicting t8 and t11. But this situation cannot happen from the marking $\mu_{0,1}$, which can only lead to $\mu_{0,9}$, from which the PN reaches a stop after firing either ⟨t8, t9⟩ or ⟨t10, t11,t12⟩.

Figure 11.15 gives the marking graph of the previous PN (marking are denoted by the places holding a token). Any path in that graph from node $\mu_{0,1}$ to $\mu_{0,9}$ is possible, hence markings such as $\mu_{0,3,8}$ or $\mu_{0,4,7}$ may or may not be reached. This marking graph is acyclic: it stops at μ_0. But in general marking graphs are cyclic. □

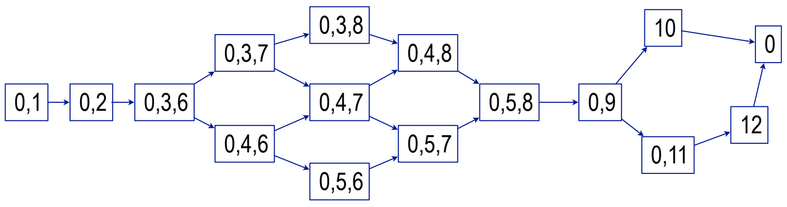


Figure 11.15. Marking graph of the PN in Figure 11.14. A node is a marking μ labeled with the subset of places that hold a token.

In summary, a PN models activities and events with simple sequences ⟨*place, transition, place*, ...⟩ and with:

1. fork transitions followed by concurrent processes, e.g., the fork t2 in previous example,
2. join transitions, synchronizing processes, e.g., the join t7,
3. nondeterministic choices between follow up processes, e.g., the between t8 and t10 in p9,
4. disjunctions of events that may lead to a state, e.g., t9 or t12 for μ_0.

In some cases a place does not represent an entire process, but possibly its beginning, its end or a logical condition, such as p0 that ends the PN activity. For that reason transition with two successor places may be a particular fork. For example, if we add in the PN of Figure 11.14 two edges (t9, p1) and (t11, p1), the process goes back to its initial marking $\mu_{0,1}$ once it reaches its marking μ_0, instead of stopping.

11.4.2 An Acting Engine for Petri Nets

The basic PN interpretation described above defines a state transition system with *instantaneous* transitions from a marking μ to a marking $\gamma(\mu, t)$. It does not distinguish between a place just receiving a token and place ready to transfer the token to the next marking. Let us see how to extend this interpretation.

Consider a binary PN where transitions are instantaneous events, e.g., the starting of an action, and places model durative activities, e.g., performing an action, computing, testing a condition, reading a sensor. The activity starts when a place p receives a token. When the activity finishes, this is signaled by some mean, e.g., a flag, telling that p is *ready* to pass over the token to the next marking. It may stay ready for a while until all other places in $In(t)$ are also ready and allow the firing of t.

This mechanism is easily extended to nonbinary PN. It can be implemented by revising the firing rules *(i)* to *(iii)* with the notion of a *firable* transition conditioned of the transition being *enabled* and its parents *ready*. We add the rule:

(i)' a transition t is *firable* if it is enabled and all the places in *In(t)* are *ready*.

Only firable transition can be fired: $\gamma(\mu, t) = \mathsf{nil}$ unless t is firable in μ.

PNAE is a simple nondeterministic algorithm for acting with a PN. If there are no enabled transitions, the PN has reached a termination state or a deadlock (e.g., the marking μ_0 of Figure 11.14). PNAE returns the current marking from which the termination or deadlock can be identified. If there are enabled but no firable transitions, then one transition may eventually become firable when all its input places are ready; PNAE loops. The Select procedure in step 2 has to distinguish conflicting from non-conflicting transitions. If t is not conflicting with any other transition in *Firable*, then firing t does not reduce *Enabled* nor *Firable*; the other firable transitions will be fired in the following iterations. If t is conflicting, there is an exclusive and critical choice of a transition in the conflict set to fire.

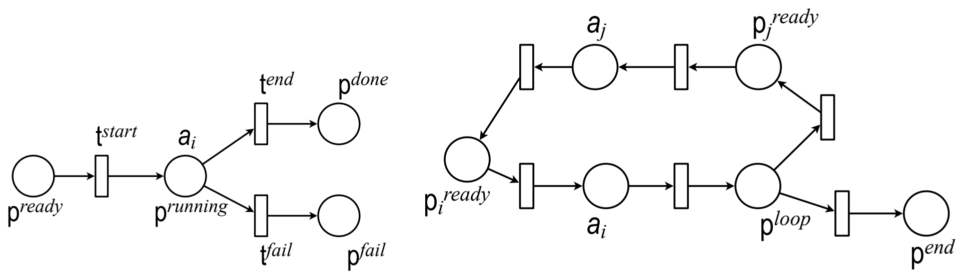
```
PNAE($\mu$)
    while True do
        $Enabled \leftarrow \{t \in Transitions \mid t$ is enabled in $\mu\}$
1       if $Enabled = \varnothing$ then return $\mu$
        $Firable \leftarrow \{t \in Enabled \mid t$ is firable$\}$
        if $Firable \neq \varnothing$ then
2           $t \leftarrow$ Select($Firable$)
            $\mu \leftarrow \gamma(\mu, t)$                          // t is fired
```

Algorithm 11.3. A Petri Net Acting Engine, PNAE.

In a sequence of firings, PNAE may trigger concurrent activities in different places, but not simultaneously. For example, after firing t2 in Figure 11.14, the algorithm pursues either in p3 or p6, while both may start running simultaneously. In other words, there is no edge $(0, 3, 6)$ to $(0, 4, 7)$ in the marking graph of Figure 11.15. Often, the difference between starting sequentially or simultaneously concurrent subnets may not be important, since firing is supposed to be instantaneous and PNEA loop is very simple. However in other cases, sequential firing may not lead to concurrent activities in parallel places. In some cases, one may want to allow concurrent branches to start simultaneously when there are no synchronization constraint. This can be emulated with PNAE by ordering *Firable* such that whenever possible concurrent branches are progressed alternatively, following a "fairness" principle. A dynamic ordering of *Firable* at each iteration is preferable to the fixed ordering of all transitions (as usually assumed in PN literature), but it requires some scheduling mechanism. This emulation of simultaneous concurrent triggering by ordering transitions is however rather constrained in distributed systems. A concurrent version of PNAE allowing non-conflicting transitions to trigger simultaneously may be preferable (see Exercise 11.8).

11.4.3 Modeling an Acting Domain with Petri Nets

Modeling an action. To model a primitive action a_i with a PN, we need at least two transitions t_i^{start} and t_i^{end} separated by a place $\mathsf{p}_i^{running}$. The latter holds a token as long as a_i is executing; it becomes ready when execution finishes. It is also convenient to have a place before t_i^{start} and a place after t_i^{end} in order to link a_i with whatever takes place before and after a_i. In general, a_i may succeed or fail. This can be modeled with a nondeterministic choice in $\mathsf{p}_i^{running}$ (see Figure 11.16(a)). The branching over the two successor transitions is conditioned on a value returned when a_i finishes. The same construct can be used to model different outcomes of a_i, e.g., for a sensing action.



(a) (b)

Figure 11.16. (a) A PN model for an action with two possible outcomes; (b) A PN model for a loop over two actions.

Combining actions into tasks. A sequence of two actions $\langle a_i, a_j \rangle$ can be modeled by merging p_i^{done} with p_j^{ready}: the transition t_j^{start} can be triggered as soon as p_i^{done} has a token, meaning that a_i finishes and a_j starts. To model the concurrent execution of a_i and a_j, we use a fork transition with two successors p_i^{ready} and p_j^{ready}. A conditional execution of an action is modeled with a place followed by a nondeterministic choice. This is illustrated in Figure 11.16(b) for modeling a loop:

here a_j follows a_i conditioned on the branching in p^{loop} which tests some external condition and either loops over a_j or exits to p^{end}.

Composition of PNs. The composition of several PNs representing individual tasks is an essential operation for a modular design. PNs can be composed in different ways.

A composition, called *PN chaining*, merges two places from two PNs, as we did for the sequence of two actions. These merged places have to meet a few conditions allowing for a proper dynamic of the composed network. For example, one could merge the place p^{end} in Figure 11.16(b) with a place p^{ready} of a sequence of actions.

Another type of PN composition, called *synchronous products*, merges transitions. This is particularly convenient for composing concurrent activities by merging their triggering transitions into a fork, as illustrated in Example 11.20.

More complex compositions can be performed through the simultaneous merging of several pairs of places and/or transitions of two networks.

Let us illustrate with a simple example composition of PN by merging transition.

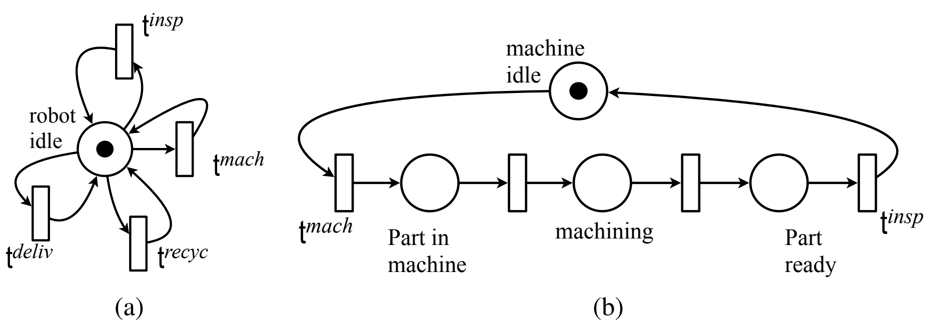


Figure 11.17. (a) A PN model of the activity performed by the robot; (b) a PN model of the machining cycle.

Example 11.20. Here, we extend VSR domain of Example 11.17. The robot has to serve a simple jobshop by bringing parts from an entry stock to a machining station, then an assembly and inspection station, then to a delivery dock or to a recycling bin, depending on the result of the inspection of the part. Four PNs model the domain.

At the highest level, a PN has one place, stating that the robot is free, and four transitions models the robot's activity (Figure 11.17(a)) :

- t^{mach}: the robot takes a part from the entry stock to the machining station,
- t^{insp}: it takes a machined part to the assembly and inspection station,
- t^{deliv} : it takes a good assembled part to the delivery dock,
- t^{recyc}: it takes a faulty assembled part to the recycling bin.

Each of these transitions correspond to a task to be hierarchically refined into subnets (see Exercise 11.6). Other places will be added as predecessors of these transitions to condition their firing in a varying and context depend order.

Transition t^{mach} requires to have parts available in the entry stock and the machining station to be ready to take a new part. These two conditions are modeled with two PNs showed respectively in Figure 11.18(a) and Figure 11.17(b). The first one assumes an infinite number of parts in the entry stock; a more realistic model would use a nonbinary PN and handle the replenishing of the stock. The second PN models the machining action and loops through t^{insp} to an additional place in $In(\mathsf{t}^{mach})$.

The PN in Figure 11.18(b) models similarly the assembly and inspection stage with a branching conditioned on whether the finished part is good or faulty; the corresponding places are connected respectively to t^{deliv} and t^{recycl}.

The global PN for this example is obtained by merging their four identically named transitions: $\mathsf{t}^{mach}, \mathsf{t}^{inspec}, \mathsf{t}^{deliv}$ and t^{recyc} (see Exercise 11.5). The next modeling step is to refine hierarchically each of these four transitions into a more detailed PN that describes what the robot does (see Exercise 11.6). □

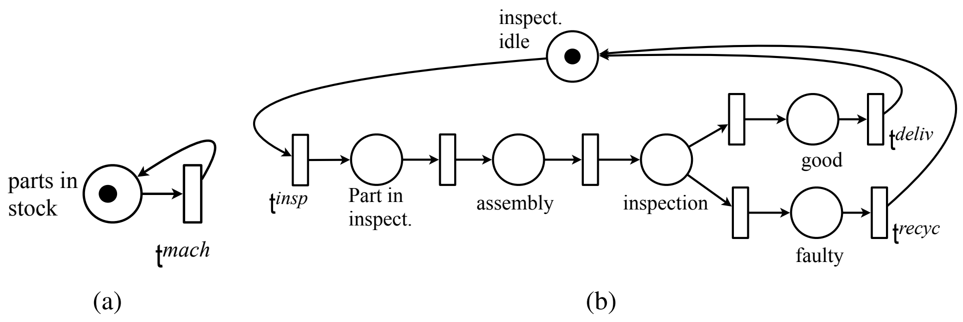


Figure 11.18. (a) A PN model of the entry stock ; (b) a PN model of the assembly and inspection cycle.

Hierarchical PNs. *Hierarchization* is another important operation for a modular specification of a domain. A hierarchical Petri Net can be defined by expanding a place into a subnetwork starting and finishing with a place. For example the place $\mathsf{p}^{running}$ in Figure 11.16(a) can be expanded as the network in Figure 11.16(b): $\mathsf{p}^{running}$ may eventually become ready when a token in Figure 11.16(b) reaches p^{end}. Not every place can be expanded into any subnetwork, e.g., p^{loop} can be expanded only in a network that allows for a choice of pursuing or finishing the loop. Similarly, a high level transition can be refined, as for a task, into a subnetwork starting and finishing with a transition.

11.4.4 Verification and Validation of Petri Nets

The analysis of a Petri Net $\mathcal{G}$ seeks to prove various correctness properties of a system modeled by $\mathcal{G}$. Let $\langle \mu_0, \mu_1, \ldots \rangle$ be a sequence of markings corresponding to a sequence of legal firings of transitions in $\mathcal{G}$. For a given application, the following questions can be of interest:

- is there a sequence $\langle \mu_0, \ldots, \mu_g \rangle$ that leads from μ_0 to a goal marking μ_g? Is $\mathcal{G}$ re-initializable, i.e., can $\mathcal{G}$ be brought from any marking μ back to μ_0?

- is there a *deadlock* in $\mathcal{G}$, that is a marking from which no transition is enabled? is such deadlock reachable from μ_0?
- is there a *dead* transition t in $\mathcal{G}$ that will never be enabled from μ_0? Conversely, is t *potentially alive*, i.e., does a sequence $\langle \mu_0, \ldots, \mu \rangle$ exist such that μ enables t? Is t *alive*, i.e., potentially live for any μ reachable from μ_0?
- is $\mathcal{G}$ *bounded* starting from μ_0, i.e., is there a constant k such that no place in $\mathcal{G}$ will hold more than k tokens in any sequence starting from μ_0? Is $\mathcal{G}$ always bounded?

When $\mathcal{G}$ is always bounded then its state space is finite. Note that a binary PN is bounded by definition; its transition graph is of size in $O(2^{|Places|})$.

There are two main classes of methods for analyzing PN: *structural analysis* techniques and *reachability analysis* techniques. Structural analysis focuses on the 'structure' or topological properties of the PN graph. These are usually formulated as place invariance or transition invariance properties. Structural analysis techniques are computationally fast but never complete: they can fail to prove a true property of a PN. Reachability analysis explores (part of) the state space, i.e., the marking graph of a PN. They are computationally expensive but complete. Implicit exploration methods, such as model checking, allow scaling up reachability analysis in many applications. Note however that correctness properties are not preserved by composition nor hierarchical refinement, i.e., proving the correctness of each PN component in a modular design does not guarantee that the composed PN is correct.

The basic PN representation considered in this section is implicitly "grounded": it refers to constant objects, values and propositions. Generalizing the domain of Example 11.20 with several robots, locations and working stations, as in Example 14.4, would demand extensive modeling for each fully grounded instance of such a domain.

The basic PN representation can be extended in different ways, for example:

- PNs with integer weights on arcs, specifying the number of tokens a transition consumes and produces when it fires (these weights replace "1" in the firing rules *(i)* and *(ii)* of Section 11.4.1);
- PNs with temporal constraints specifying an interval for the duration of an activity in a place, or how long a transition can remain enabled once its predecessors have tokens without being fired;
- PNs with typed tokens and labels on transitions, conditioning their firing to the testing an external predicate.
- *Colored Petri Nets* where tokens are programming objects of different classes permitting various computational operations in places that hold them. Colored PN are Turing complete.

Conversely, we may restrict the basic PN model in different ways, for example:

- PN where every transition is preceded and followed by only one place, that is $|In(t)| = |Out(t)| = 1$. This class of PN, called state machines, has the expressiveness of finite state automata
- PN where every place is preceded and followed by at most one transition. This class, called marked graphs, exclude nondeterministic choices.

The analysis of these restricted classes is simpler, but their expressiveness is limited.

An extensive literature discusses the properties of the mentioned types of PNs (see next section). For our purpose here, the important point to underline is that the formalism of PN is very rich, with the following caveats:

- Modeling a domain with PN requires significant engineering efforts. There are only a few contribution to the synthesis of PN models. To our knowledge, there is no easy way for mapping a factored representation of a state space into a PN whose marking graph represents such a state space. Nor is there conversely an easy way of going from a PN to the factored representation it represents. One has to specify the two in parallel and make sure that they map consistently.
- The correctness analysis of PN is a significant advantage for the development of safety critical applications; it pays for the modeling effort. Tools such as TINA [123] or CPN [549], allow for a modular design and perform quite efficiently their analysis on the composed PN (since correctness is not preserved by composition). Generalized PN are more complex to analyze.

11.5 Discussion and Bibliographic Notes

Finite State Automata. FSA have been used as acting models in which a primitive action is represented as an FSA whose transitions are labelled with sensory-motor signals and commands. For example, FSA have been used jointly with a temporal planner IxTeT [219]. PLEXIL illustrates a FSA representation in which nodes are computational abstractions [1136]. A node can monitor events, execute commands, or assign values to variables. It may refer hierarchically to a list of lower level nodes. Execution is controlled by constraints (start, end), guards (invariant), and conditions. SMACH [150], the execution system of the Robot Operating System, ROS, also implements an automata-based approach. The user writes a set of hierarchical state machines. Each state corresponds to the execution of a particular command. The interface with ROS actions, services, and topics is very natural, but the semantics of constructs available in SMACH is limited for reasoning on goals and states.

There is a vast literature on the specification, verification, and synthesis of automata, including works on automata theory. For a nice introduction to automata theoretic approach to linear temporal logic see [1116]. The techniques presented in Section 11.2 for acting with input/output automata are based on the work on planning with asynchronous processes, which have has been first proposed in [902] and then formalized and extensively evaluated in [129]. Such techniques have been extended to deal with service oriented applications in [188].

Behavior Trees. BTs expressiveness is equivalent to that of finite state automata, but BTs have been found more convenient for supporting task hierarchy, reactivity to events, and modular design. They were first introduced for computer gaming to control automated opponents [544], and applied in the video games industry [762, 365]. They have been further developed in other areas, notably in robotics, and extended with planning and learning capabilities (see Chapter 11). A comprehensive coverage of

BT is given in [252] and the survey [542]. Several development tools and BT libraries are listed in [542]. BTs have been extended for parallel execution [250], runtime verification [254], and for handling concurrency [251].

Acting systems with BT are becoming popular, particularly in robotics. For example, the open robotics navigation package NAV2 (a successor of the navigation stack of ROS) supports BT for reconfiguration and adaptation to the specifics of a robotics application.[14] Other systems uses BT as an acting system for ground robots, e.g., [757, 876], aerial robots [678, 206], or underwater robots [1047].

Petri Nets. PNs have been proposed in the early sixties by Petri [893] as a representation for studying communicating automata. They have been widely developed as a formal model for the specification and analysis of event based systems, convenient for their ability to express concurrency and synchronization with constraints, such as conflicts and mutual exclusion.

The basic PN model was extended in may directions, e.g., transitions with priorities and/or timing constraints, typed tokens, labels depicting conditions on places and tokens (colored PN), and probabilistic firing delays on transitions (stochastic PN). Research on PN is very active, with a yearly conference [117], and numerous surveys and text books, e.g., [892, 548, 296]. Example 11.20 is inspired from [204]. The automated synthesis of PN models has received a few contributions, e.g., [72].

PN developments have been associated with several software tools to support the specification, modeling and analysis of systems. The Petri Nets community maintains a repository of references and services,[15] including an extensive database of software packages, among which TINA [123] and CPN [549] offer extensive verification capabilities for timed and colored PN.

In the context of acting and planning, concurrency and synchronization issues very frequently arise at the primitive action level. They have been addressed with PNs in a several approaches, e.g., to model the proper order of the execution of actions and their required coordination [1148]. The model can be used in simulation for verification and performance testing. ASPiC [696] is an acting system which models robot's skills (i.e., primitive actions) using a specific colored PN representation [911]; it offers some soundness and safety verification capabilities. Other PN-based approaches have been pursued, e.g., to specify acting systems whose properties can be validated with reachability and deadlock analysis [82, 1237]. Planning techniques based on Petri Nets are proposed in e.g., [166, 1237].

11.6 Exercises

11.1. Can all (memoryless) policies be written as contingent plans, that is, plans with conditional tests? Vice versa? Explain the answer with some examples.

11.2. Consider Figure 11.19.

[14] See https://www.behaviortree.dev/

[15] see https://www.informatik.uni-hamburg.de/TGI/PetriNets/index.php

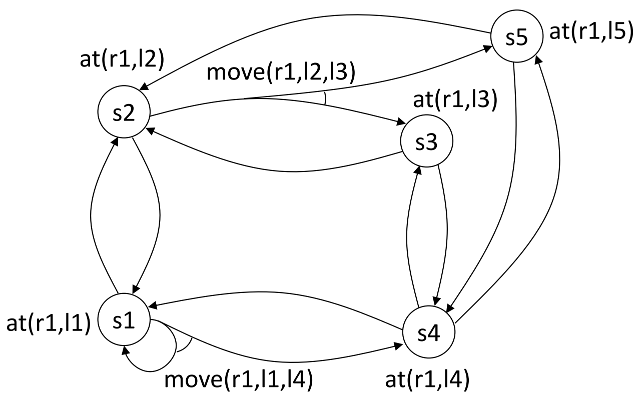


Figure 11.19. A nondeterministic state-transition system.

(a) Give an example of an unsafe solution π_1, a cyclic safe solution π_2 and an acyclic safe solution π_3 to the problem of moving from s1 to s5, if one exists. Draw their reachability graphs, circling the leaves.

(b) Suppose the initial state was s2 instead of s1. Are π_1, π_2, and π_3 solutions? If so, what kinds?

11.3. Prove that a policy π is an unsafe solution iff

$$\exists s \in \mathit{leaves}(s_0, \pi) \text{ s.t. } s \notin S_g \vee \exists s \in \widehat{\gamma}(s_0, \pi) \text{ s.t. } \mathit{leaves}(s, \pi) = \varnothing$$

11.4. Rewrite the definitions of safe cyclic and acyclic solutions (see Section 11.1.2) to ensure that these solutions reach the goal and then continue looping within the set of goal states. Additionally, define solutions that traverse the set of goal states infinitely often.

11.5. Define the global PN obtained by composing the four nets given in Example 11.20.

11.6. Define a PN for refining each of the four transitions of Example 11.20:

- t^{mach}: the robot goes to the entry stock, takes a part and brings it to the machining station,
- t^{insp}: it goes to the machining station, takes a machined part and brings the assembly and inspection station,
- t^{deliv} : it goes to the inspection station, takes a good assembled part and brings it to the delivery dock,
- t^{recyc}: it goes to the inspection station, takes a faulty assembled part and brings it to the recycling bin.

11.7. Is there a dead transition in the PN of Figure 11.20 from the marking shown? Does this PN have a deadlock?

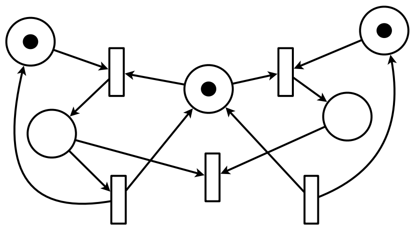


Figure 11.20. Example PN for Exercise 11.7.

11.8. This exercise is about the study of CoPNAE, a PN acting algorithm that handles concurrency. We assume that a concurrent execution in a PN starts only after the firing of a transition explicitly labelled as a *fork*. The concurrent subnets following a fork may have links to the rest of a PN, in particular they may have incoming and outgoing edges, e.g., to bring tokens from, or to constrain a synchronization with, other parts of a PN. We do however assume that the set of places and the set of transitions in each concurrent subnet are proper to that subnet, i.e., there is no overlap with the places or transitions in the rest of the PN (the fork and possibly the join transition are no within the concurrent subnet). We also assume that there are no conflicts across subnets, i.e., any conflicting pair (t, t') is either within a subnet or outside of it. This is in particular to permit a choice in a conflict that remains local in a subnet.

We associate to a fork t followed by k concurrent branches a partition of the set *Transitions* into $Fork(t) = \{Bt_1, \ldots, Bt_k, Bt_0\}$, where the Bt_i for $1 \leq i \leq k$ are the *disjoint* subsets of transitions corresponding to the concurrent subnet branches starting at t, and $Bt_0 = Transitions \setminus \cup_{i=1,k} BT_i$. For example, if t2 in Figure 11.14 is labeled as a fork, then $Fork(\text{t2}) = \{\{\text{t3, t4}\}, \{\text{t5, t6}\}, Transitions \setminus \{\text{t3, t4, t5, t6}\}\}$,

```
CoPNAE($\mu$, $Transitions$)
    while True do
        $Enabled \leftarrow \{t \in Transitions$ — $t$ is enabled in $\mu\}$
1       if Enabled $=\varnothing$ then return($\mu$)
        $Firable \leftarrow \{t \in Enabled$ — $t$ is firable$\}$
2       $t \leftarrow$ Guide $(Firable)$
        $\mu \leftarrow \gamma(\mu, t)$                                    // t is fired
        if $t$ is a fork then
            $Mu \leftarrow \varnothing$
            for each $Bt \in Fork(t)$ concurrently do
3               $Mu \leftarrow Mu \cup \{$CoPNAE$(\mu, Bt)\}$
4           $\mu \leftarrow$ Update$(\mu, Mu)$
```

Algorithm 11.4. A Concurrent Petri Net Acting Engine, CoPNAE

After firing a fork transition t, algorithm CoPNAE (Algorithm 11.4) pursue concurrently the execution in every concurrent subnet following t, as well as in the remaining part of the network, focusing on the transitions specific to each component of the partition *Fork*(t). *Mu* is the set of firings returned by the concurrent calls. It used by Update to compute a marking μ identical to the one obtained by advancing sequentially on the components of *Fork*(t), assuming that conflicts, if any, would be solved in identical ways in the concurrent as well as the sequential cases.

Prove that Update can be expressed in the vector notation of markings as:

$$\sum_{\mu_i \in Mu} \mu_i - k \times \mu$$

(Here μ is a vector of $|Places|$ integer components, with the usual sum and scalar multiplication over vectors)

Run CoPNAE on the PN of Figure 11.14. Compare with PNAE.

11.9. Let v be an interior BT node with $Children(v) = \langle v_1, \ldots, v_n \rangle$, and suppose $\text{exec-status}(v_i) \in \{\text{Success}, \text{Failure}\}$ for every $v_i \in Children(v)$.

(a) Suppose we assign the following numeric values to execution statuses: Success = 1, and Running = Failure = 0. Prove that if $type(v)$ = And, then

$$\text{exec-status}(v) \equiv \min(\text{exec-status}(v_1), \ldots, \text{exec-status}(v_n)). \tag{11.1}$$

(b) Suppose we assign the following numeric values to execution statuses: Success = Running = 1, and Failure = 0. Prove that if $type(v)$ = Or, then

$$\text{exec-status}(v) \equiv \max(\text{exec-status}(v_1), \ldots, (\text{exec-status}(v_n)). \tag{11.2}$$

(c) Suppose we assign the following numeric values to execution statuses: Success = 1, Running = 0.5, and Failure = 0. Then do (11.1) and (11.2) still hold? Why or why not?

11.10. Develop the behavior tree in Figure 11.12 the handle the case where the condition hold(r)=free does not hold by having the robot move it needed to the recycling bin and putting what it holds there.

12 Planning with Nondeterministic Models

In this Chapter, we propose different approaches to planning with nondeterministic models. We describe three techniques for planning with nondeterministic state transition systems: And/Or graph search (Section 12.1), planning based on on determinization techniques (Section 12.2), and planning via symbolic model checking (Section 12.3). We then present techniques for planning by synthesis of input/output automata (Section 12.4). We finally discuss briefly techniques for Behavior Trees generation (Section 12.5).

12.1 And/Or Graph Search

A nondeterministic model can be represented as an And/Or graph (see Appendix A) in which each Or-branch corresponds to a choice among the actions that are applicable in a state, and each And-branch corresponds to the possible outcomes of the chosen action. In this section, we present algorithms that search And/Or graphs to find solutions.

```
Find-Solution $(\Sigma, s_0, S_g)$
    $\pi \leftarrow \varnothing$;  $s \leftarrow s_0$;  $Visited \leftarrow \{s_0\}$
    loop
        if $s \in S_g$ then return $\pi$
        $A' \leftarrow Applicable(s)$
        if $A' = \varnothing$ then return failure
        **nondeterministically choose** $a \in A'$
        **nondeterministically choose** $s' \in \gamma(s, a)$
        if $s' \in Visited$ then return failure
        $\pi(s) \leftarrow a$;  $Visited \leftarrow Visited \cup \{s'\}$; $s \leftarrow s'$
```

Algorithm 12.1. Planning for solutions by forward search.

We first present a simple algorithm that finds a solution by searching the And/Or graph forward from the initial state. Find-Solution (see Algorithm 12.1) is guaranteed to find a solution if it exists. The solution may be either safe or unsafe. It is a simple modification of forward state-space search algorithms for deterministic models (see Section 3.1). The main point related to nondeterminism is in the "progression" line (**nondeterministically choose** $s' \in \gamma(s, a)$), where we nondeterministically search for all possible states generated by $\gamma(s, a)$. Find-Solution simply searches the And/Or

Free pre-publication, for personal use only. To be published by Cambridge University Press.

graph to find a path that reaches the goal, without keeping track of which states are generated by which action. In this way, Find-Solution ignores the real complexity of nondeterminism in the model. It deals with the And-nodes as if they were Or-nodes, that is, as if it could choose which outcome would be produced by each action.

Recall that the nondeterministic choices "**nondeterministically choose** $a \in A'$" and "**nondeterministically choose** $s' \in \gamma(s, a)$" correspond to an abstraction for ignoring the precise order in which the algorithm tries actions a among all the applicable actions to state s and alternative states s' among the states resulting from performing a in s.

Example 12.1. Consider the problem described in Example 11.1. Let the initial state s_0 be on_ship, and let the set of goal states S_g be $\{$gate1, gate2$\}$. Find-Solution proceeds forward from the initial state on_ship. It finds initially only one applicable action, that is, unload. It then expands it into at_harbor, one of the possible nondeterministic choices is s' = parking1, which gets then expanded to gate2; π_1 (see Example 11.2) is generated in one of the possible nondeterministic execution traces. □

```
Find-Safe-Solution $(\Sigma, s_0, S_g)$
    $\pi \leftarrow \varnothing$
    $Frontier \leftarrow \{s_0\}$
    for every $s \in Frontier \setminus S_g$ do
        $Frontier \leftarrow Frontier \setminus \{s\}$
        if $Applicable(s) = \varnothing$ then return failure
        nondeterministically choose $a \in Applicable(s)$
        $\pi \leftarrow \pi \cup (s, a)$
        $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus Domain(\pi))$
        if has-unsafe-loops$(\pi, a, Frontier)$ then return failure
    return $\pi$
```

Algorithm 12.2. Planning for safe solutions by forward search.

Algorithm 12.2 Find-Safe-Solution is a simple algorithm that finds safe solutions. The algorithm performs a forward search and terminates when all the states in *Frontier* are goal states. Find-Safe-Solution fails if the last action introduces a "bad loop", that is, a state from which no state in *Frontier* is reachable. The routine has-unsafe-loops checks whether a "bad loop" is introduced. A "bad loop" is introduced when the set of states resulting from performing action a, which are not in the domain of π, will never lead to the frontier:

$$\text{has-unsafe-loops}(\pi, a, Frontier) \text{ iff}$$
$$\exists s \in (\gamma(s, a) \cap Domain(\pi)) \text{ such that } \hat{\gamma}(s, \pi) \cap Frontier = \varnothing.$$

Algorithm 12.3 Find-Acyclic-Solution is a simple algorithm that finds safe acyclic solutions. The algorithm is the same as Find-Safe-Solution, but in the failure condition.

```
Find-Acyclic-Solution $(\Sigma, s_0, S_g)$
    $\pi \leftarrow \varnothing$
    $Frontier \leftarrow \{s_0\}$
    for every $s \in Frontier \setminus S_g$ do
        $Frontier \leftarrow Frontier \setminus \{s\}$
        if $Applicable(s) = \varnothing$ then return failure
        nondeterministically choose $a \in Applicable(s)$
        $\pi \leftarrow \pi \cup (s, a)$
        $Frontier \leftarrow Frontier \cup (\gamma(s, a) \setminus Domain(\pi))$
        if has-loops$(\pi, a, Frontier)$ then return failure
    return $\pi$
```

Algorithm 12.3. Planning for safe acyclic solutions by forward search.

It fails if the last action introduces a loop, that is, a state from which the state itself is reachable by performing the plan:

$$\begin{aligned}&\text{has-loops}(\pi, a, Frontier) \text{ iff}\\ &\quad \exists s \in (\gamma(s, a) \cap Domain(\pi)) \text{ such that } s \in \hat{\gamma}(s, \pi)\end{aligned}$$

Example 12.2. Consider the problem P with nondeterministic model Σ described in Example 11.1, initial state on_ship, and set of goal states S_g as {gate1, gate2}. Find-Acyclic-Solution starts from the initial state on_ship, for every state s in the frontier expands the frontier by performing $\gamma(s, a)$. A successful trace of execution evolves as follows[1]:

$$\begin{aligned}\text{Step}_0 &: \ \text{on\_ship}\\ \text{Step}_1 &: \ \text{at\_harbor}\\ \text{Step}_2 &: \ \text{parking2, parking1, transit1}\\ \text{Step}_3 &: \ \text{transit3, gate1, gate2, transit2}\\ \text{Step}_4 &: \ \text{gate1, gate2}\end{aligned}$$

□

We introduce now a technique that is based on a cost model of actions. Recall cost models defined in Section 2.2. We assign a cost to each action that is performed in a state, $cost(s, a)$. Weighting actions with cost can be useful in some application domains, where, for instance, actions consume resources or are more or less difficult or expensive to perform. Algorithm 12.4 Find-Acyclic-Solution-by-MinMax uses costs to identify which may be the best direction to take. It starts from the initial state and selects actions with minimal costs among the ones that are applicable. We are interested in finding a solution with the minimum accumulated cost, that is, the minimum of the costs of each action that is selected in the search. Because the domain model is nondeterministic and $\gamma(s, a)$ results in different states, we want to minimize

[1] For simplicity, in the following we use on_ship, at_harbor, and so on, as names of states rather than a state variable notation.

```
    Find-Acyclic-Solution-by-MinMax ($\Sigma$,$S_0$,$S_g$)
        return Compute-worst-case-for-action($S_0$, $S_g$, $\infty$, $\varnothing$)

Compute-worst-case-for-action($S$, $S_g$, $\beta$, ancestors)
        $c' \leftarrow -\infty$
        $\pi' \leftarrow \varnothing$
        // if S is nonempty, this loop will be executed at least once:
        for every $s \in S$
            if $s \in$ ancestors then
                return ($\pi'$,$\infty$)
            ($\pi$,$c$) $\leftarrow$ Choose-best-action($s$, $S_g$, $\beta$, ancestors $\cup\{s\}$)
            $\pi' \leftarrow \pi \cup \pi'$
            $c' \leftarrow \max(c', c)$
            if $c' \geq \beta$ then
                break
        return ($\pi'$, $c'$)
```

Algorithm 12.4. Planning for safe acyclic solutions by MinMax Search.

the *worst-case* accumulated cost, that is, the maximum accumulated cost of each of the possible states in $\gamma(s,a)$. This is given by the following recursive formula:

$$c(s) = \begin{cases} 0 & \text{if } s \text{ is a goal,} \\ \min_{a \in Applicable(s)} (\text{cost}(a) + \max_{s' \in \gamma(s,a)} c(s')) & \text{otherwise.} \end{cases}$$

For this reason, the algorithm is said to perform a “MinMax search.” While performing the search, the costs of actions that are used to expand the next states are accumulated, and the algorithm checks whether the accumulated cost becomes too high with respect to alternative selections of different actions. In this way, the accumulated cost is used to find an upper bound in the forward iteration.

Find-Acyclic-Solution-by-MinMax (Algorithm 12.4) finds safe acyclic solutions for nondeterministic models that may have cycles. It returns a pair (π,c), where π is a safe acyclic solution that is *worst-case optimal*, that is, the maximum cost of executing π is as low as possible, and c is the maximum cost of executing π.

Find-Acyclic-Solution-by-MinMax implements a depth-first search by minimizing the maximum sum of the costs of actions along the search. It alternates recursively between calls to Choose-best-action (Algorithm 12.5) and Compute-worst-case-for-action. The former calls the latter on the set of states $\gamma(s,a)$ resulting from the application of actions a that are applicable to the current state s, where Compute-worst-case-for-action returns the policy π' and its corresponding cost c'. Visited states are accumulated in the “ancestors” variable. Choose-best-action then updates the cost of π with the cost of the action ($c = c' + cost(s,a)$), and updates the policy with the selected action in the current state ($\pi = \pi' \cup (s,a)$). In the Choose-best-action procedure, β keeps track of the minimum cost of alternative policies computed at each iteration, which is compared with the maximum cost computed over paths in π

```
Choose-best-action($s$, $S_g$, $\beta$, ancestors)
    if $s \in S_g$ then
        return ($\varnothing$, 0)
    else if $Applicable(s) = \varnothing$ then
        return ($\varnothing$, $\infty$)
    else do
        $c = \infty$
        // this loop will always be executed at least once:
        for every $a \in$ Applicable($s$) do
            $(\pi', c') \leftarrow$ Compute-worst-case-for-action($\gamma(s, a)$, $S_g$, $\beta$, ancestors)
            if $c > c' + cost(s, a)$ then do
                $c \leftarrow c' + cost(s, a)$
                $\pi(s) \leftarrow a$
            $\beta \leftarrow min(\beta, c)$
        return ($\pi$, $c$)
```

Algorithm 12.5. The policy with minimal cost over actions.

by Compute-worst-case-for-action (see the instruction $c' = max(c', c)$). If the current children's maximum cost c' is greater than or equal to the current minimum cost β, then the policy π' gets discarded and control gets back to Choose-best-action which chooses a different action.

Indeed, while we are considering each state $s' \in \gamma(s, a)$, the worst-case cost of a policy that includes an action a is greater than the maximum cost at each s' visited so far. We know that elsewhere in the And/Or graph there exists a policy whose worst case cost is less than β. If the worst-case cost of a policy that includes a is greater or equal to β, then we can discard a.

Find-Acyclic-Solution-by-MinMax's memory requirement is linear in the length of the longest path from s_0 to a goal state, and its running time is linear in the number of paths from s_0 to a goal state.

Find-Acyclic-Solution-by-MinMax ignores the possibility of multiple paths to the same state. If it comes to a state s again along a different path, it does exactly the same search below s that it did before. One could use memoization techniques to store these values rather than recomputing them – which would produce better running time but would require exponentially more memory. See Exercise 12.5.

12.2 Determinization Techniques

Determinization techniques address the problem of planning with nondeterministic models by *determinizing the nondeterministic model*. Intuitively the idea is to consider one of the possible many outcomes of a nondeterministic action at a time, using an efficient classical planning technique to find a plan that works in the deterministic case. Then different nondeterministic outcomes of an action are considered and a new plan for that state is computed, and finally the results are joined in a contingent plan that considers all the possible outcomes of actions. Of course, it may be that when a partial plan is extended to consider new outcomes, no solution is possible, and the algorithm must find an alternative solution with different actions.

12.2.1 Guided Planning for Safe Solutions

Before getting into the details, we show a basic idea underlying determinization techniques. Safe solutions can be found by starting to look for (possibly unsafe) solutions, that is, plans that may achieve the goal but may also be trapped in states where no action can be executed or in cycles where there is no possibility of termination. The idea here is that finding possibly unsafe solutions is much easier than finding safe solutions. Compare indeed the algorithm for finding solutions Find-Solution and the one for finding safe solutions Find-Safe-Solution in Section 12.1. Whereas Find-Solution does not distinguish between And-branches and Or-branches, Find-Safe-Solution needs to check that there are no unsafe loops, and this is done with the has-unsafe-loops routine.

```
Guided-Find-Safe-Solution ($\Sigma$,$s_0$,$S_g$)
    if $s_0 \in S_g$ then return($\varnothing$)
    if $Applicable(s_0) = \varnothing$ then return(failure)
    $\pi \leftarrow \varnothing$
    loop
        $Q \leftarrow leaves(s_0, \pi) \setminus S_g$
        if $Q = \varnothing$ then do
            $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
            return($\pi$)
        select arbitrarily $s \in Q$
        $\pi' \leftarrow$ Find-Solution($\Sigma, s, S_g$)
        if $\pi' \neq$ failure then do
            $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin Domain(\pi)\}$
        else for every $s'$ and $a$ such that $s \in \gamma(s', a)$ do
            $\pi \leftarrow \pi \setminus \{(s', a)\}$
            make $a$ not applicable in $s'$
```

Algorithm 12.6. Guided planning for safe solutions.

Algorithm 12.6 Guided-Find-Safe-Solution is based on this idea, that is, finding safe solutions by starting from possibly unsafe solutions that are found by Find-Solution. Guided-Find-Safe-Solution takes in as input a problem in a nondeterministic model Σ with initial state s_0 and goal states S_g. If a safe solution exists, it returns the safe solution π. The algorithm checks first whether there are no applicable actions in s_0. If this is the case, it returns failure. In the loop, Q is the set of all nongoal leaf states reached by π from the initial state. If there are no nongoal leaf states, then π is a safe solution. When we have the solution, we get rid of the part of π whose

states are not reachable from any of the initial state (we say we "clean" the policy). If there are instead nongoal leaf states reached by π, then we have to go on with the loop. We select arbitrarily one of the nongoal leaf states, say, s, and find a (possibly unsafe) solution from initial state s with the routine Find-Solution, see Algorithm 12.1. If Find-Solution does not return failure, then π' is a (possibly unsafe) solution, and therefore we add to the current policy π all the pairs (s, a) of the (possibly unsafe) solution π' that do not have already a state s in π. If a (possibly unsafe) solution does not exists (the else part of the conditional) this means we are trapped in a loop or a dead end without any possibility of getting out. We therefore get rid from π of all the pairs (s', a) that lead to dead-end state s. We implement this by making action a not applicable in s'.[2] In this way, at the next loop iteration, we will not have the possibility to become stuck in the dead end.

12.2.2 Planning for Safe Solutions by Determinization

The idea underlying the Guided-Find-Safe-Solution algorithm is to use possibly-unsafe solutions to find safe solutions. Find-Solution returns a path to the goal by considering only one of the many possible outcomes of an action. Looking for just one action outcome and finding paths inspires the idea of determinization. If we replace each action a leading from state s to n states $s_1, \ldots, s_n$ with n deterministic actions $a_1, \ldots, a_n$, each one leading to a single state $s_1, \ldots, s_n$, we obtain a deterministic model, and we can use classical efficient planners to find solutions in the nonderministic model as sequences of actions in the deterministic model. We will have then to transform a sequential plan into a corresponding policy, and to extend it to consider multiple action outcomes. According to this idea, we define a determinization of a nondeterministic model.[3]

Algorithm 12.7 exploits model determinization and replaces Find-Solution in Guided-Find-Safe-Solution with search in a deterministic model. Here we use the simple forward search algorithm Forward-Search presented in Section 3.1, but we could use a more sophisticated classical planner, as long as it is complete (i.e., it finds a solution if it exists). This algorithm is similar to the first algorithm for planning by determinization proposed in literature.

Find-Safe-Solution-by-Determinization is like Guided-Find-Safe-Solution, except for the following steps:

1. **The determinization step:** We add a determinization step. The function mk-deterministic returns a determinization of a nondeterministic model.

[2]This operation can be done in different ways, and depends on which kind of representation we use for the domain. This operation may not be efficient depending on the implementation of Σ. For example, if our domain is represented using an explicit the action representation, then modifying the domain can take exponential time in the worst case. If we have access to the source code for Find-Solution, then a better approach is to modify it to take an additional argument that's a "nogood table", i.e., a hash table of state-action pairs that the planner should never use.

[3]The operation of transforming each nondeterministic action into a set of deterministic actions is complicated by the fact that we have to take into account that in different states the same action can lead to a set of different states. Therefore, if the set of states has exponential size with respect to the number of state variables, then this operation would generate exponentially many actions.

```
Find-Safe-Solution-by-Determinization ($\Sigma$,$s_0$,$S_g$)
    if $s_0 \in S_g$ then return($\varnothing$)
    if $Applicable(s_0) = \varnothing$ then return(failure)
    $\pi \leftarrow \varnothing$
    $\Sigma_d \leftarrow$ mk-deterministic($\Sigma$)
    loop
        $Q \leftarrow leaves(s_0, \pi) \ \setminus S_g$
        if $Q = \varnothing$ then do
            $\pi \leftarrow \pi \setminus \{(s,a) \in \pi \mid s \notin \widehat{\gamma}(s_0, \pi)\}$
            return($\pi$)
        select $s \in Q$
        $p' \leftarrow$ Forward-Search ($\Sigma_d, s, S_g$)
        if $p' \neq$ failure then do
            $\pi' \leftarrow$ Plan2policy($p', s$)
            $\pi \leftarrow \pi \cup \{(s,a) \in \pi' \mid s \notin Domain(\pi)\}$
        else for every $s'$ and $a$ such that $s \in \gamma(s', a)$ do
            $\pi \leftarrow \pi \setminus \{(s', a)\}$
            make the actions in the determinization of $a$ not applicable in $s'$
```

Algorithm 12.7. Planning for safe solutions by determinization.

```
Plan2policy($p = \langle a_1, \ldots, a_n \rangle$,$s$)
    $\pi \leftarrow \varnothing$
    for $i$ from 1 to $n$ do
        $\pi \leftarrow \pi \cup (s,$ det2nondet$(a_i))$
        $s \leftarrow \gamma_d(s, a_i)$
    return $\pi$
```

Algorithm 12.8. Transformation of a sequential plan into a corresponding policy.

2. **The classical planner step:** We apply Forward-Search on the deterministic model Σ_d rather than using Find-Solution on the nondeterministic model Σ. In general, we could apply any (efficient) classical planner.
3. **The plan2policy transformation step:** We transform the sequential plan p' found by Forward-Search into a policy (see routine Plan2policy, Algorithm 12.8), where $\gamma_d(s, a)$ is the γ of Σ_d obtained by the determinization of Σ. The routine det2nondet returns the original nondeterministic action corresponding to its determinization a_i.
4. **The action elimination step:** We modify the deterministic model Σ_d rather than the nondeterministic model Σ.

12.3 Planning via Symbolic Model Checking

The conceptually simple extension led by nondeterminism causes a practical difficulty. Because one action can lead to a set of states rather than a single state, planning algorithms that search for safe (cyclic and acyclic) solutions need to analyze all the states that may result from an action. Planning based on *symbolic model checking* attempts to overcome the difficulties of planning with nondeterministic models by working on a symbolic representation of sets of states and actions. The underlying idea is based on the following ingredients:

- Algorithms search the state space by working on sets of states, rather than single states, and on transitions from sets of states through sets of actions, rather than working separately on each of the individual transition.
- Sets of states, as well as sets of transitions, are represented as propositional formulas, and search through the state space is performed by logical transformations over propositional formulas
- Specific data structures, *Binary Decision Diagrams* (BDDs), are used for the compact representation and effective manipulation of propositional formulas

Example 12.3. In this example we give a first intuition on how a symbolic representation of sets of states can be advantageous. Consider the planning problem P with the nondeterministic model Σ described in Example 11.1, the initial state s_0 is the state labeled in Figure 11.1 as on_ship, and goal states $S_g = \{\text{gate1}, \text{gate2}\}$. The states of this simple nondterministic model can be described by a single state variable indicating the position of the item, for example, a container. The state variable pos(item) can assume values on_ship, at_harbor, parking1, parking2, transit1, transit2, transit3, gate1, and gate2.

Now let's suppose that at each position, the item can be either on the ground or on a vehicle for transportation. We would have a second variable loaded, the value of which is either on_ground or on_vehicle.

Let's also suppose that we have a variable that indicates whether a container is empty, full, or with some items inside. The model gets to 54 states.

Now, if we want to represent the set of states in which the container is ready to be loaded onto a truck, this set can be compactly represented by the formula $\text{gate1} \vee \text{gate2}$. This is a symbolic, compact representation of a set of states. Now suppose that further 10 state variables are part of the model. There may be many states in which the container is ready to be loaded onto a truck, while their representation is the same as before: $\text{gate1} \vee \text{gate2}$.

BDDs provide a way to implement the symbolic representation just introduced. A BDD is a directed acyclic graph (DAG). The terminal nodes are either "truth" or "falsity" (alternatively indicated with 0 and 1, respectively). The corresponding BDDis in Figure 12.1. □

In the rest of this section, we describe the algorithms for planning via symbolic model checking both as operation on sets of states and as the corresponding symbolic transformations on formulas.

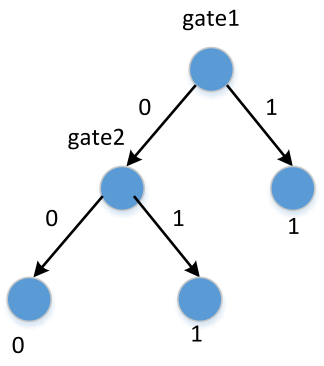


Figure 12.1. BDD for gate1 ∨ gate2

12.3.1 Symbolic Representation

A state variable representation, where each variable x_i can have a value $v_{ij} \in Range(x_i)$, can be mapped to an equivalent representation based on propositional variables. We can represent a state by means of assignments to propositional variables rather than assignments to state variables: For each state variable x_i and for each value $v_{ij} \in Range(x_i)$, we have a binary variable that is true if $x_i = v_{ij}$, and $x_i = v_{ik}$ is false for each $k \neq j$.

In symbolic model checking, a state is represented by means of propositional variables (that is, state variables that have value either true (T) or false (F)) that hold in that state. We write $P(s)$ a formula of propositional variables whose unique satisfying assignment of truth values corresponds to s. Let $\boldsymbol{x}$ be a vector of distinct propositional variables.

This representation naturally extends to any *set of states* $Q \subseteq S$. We associate a set of states with the disjunction of the formulas representing each of the states.

$$P(Q) = \bigvee_{s \in Q} P(s).$$

The satisfying assignments of $P(Q)$ are the assignments representing the states of Q.

Example 12.4. In Example 11.1, consider the case in which the item (e.g., a car) that needs to be moved to a parking area may get damaged. Moreover, the parking area can be either open or closed, and the area can be either full or have a slot where the item can be stored. We can represent the set of states of this nondeterministic model with three propositional variables in $\boldsymbol{x}$:

x_1 : status(car) = damaged
x_2 : areaavailability = open
x_3 : areacapacity= full

The set of states S of the model has eight states. The single state s_1 in which the item is not damaged, the storage area is open, and there is a slot available for storage can be represented by the assignment of truth values to the three proposition variables

$$\begin{aligned} x_1 &\leftarrow \mathsf{F} \\ x_2 &\leftarrow \mathsf{T} \\ x_3 &\leftarrow \mathsf{F} \end{aligned}$$

or analogously by the truth of the formula

$$P(s_1) = \neg x_1 \wedge x_2 \wedge \neg x_3.$$

The four states in which the car is undamaged is represented by the single variable assignment

$$x_1 \leftarrow \mathsf{F}$$

or analogously by the truth of the formula

$$P(Q) = \neg x_1.$$

□

The main effectiveness of the symbolic representation is that the cardinality of the represented set is not directly related to the size of the formula. As a further advantage, the symbolic representation can provide an easy way to ignore irrelevant information. For instance, in the previous example, notice that the formula $\neg x_1$, because it does not say anything about the truth of x_2 and x_3, represents four states, where the item is not damaged in all of them. The whole state space S (eight states) can thus be represented with the propositional formula that is always true, T, while the empty set can be represented by falsity, F. These simple examples give an intuitive idea of one of the main characteristics of a symbolic representation of states: the size of the propositional formula is not directly related to the cardinality of the set of states it represents. If we have one billion propositional variables to represent 2^{10^9} states, with a proposition of length one, for example, x, where x is one of the propositional variables of $\boldsymbol{x}$, we can represent all the states where x is true.

For these reasons, a symbolic representation can have a dramatic improvement over an explicit state representation which enumerates the states of a state transition system.

Another advantage of the symbolic representation is the natural encoding of set-theoretic transformations (e.g., union, intersection, complementation) with propositional connectives over propositional formulas, as follows:

$$\begin{aligned} P(Q_1 \cup Q_2) &= P(Q_1) \vee P(Q_2) \\ P(Q_1 \cap Q_2) &= P(Q_1) \wedge P(Q_2) \\ P(S - Q) &= P(S) \wedge \neg P(Q) \end{aligned}$$

We can use a vector of propositional variables, say $\boldsymbol{y}$, to name actions. Naming actions with a binary string of $\boldsymbol{y}$ bits will allow us to use BDDs at the implementation level in the next sections. If we have n actions, we can use $\lceil \log n \rceil$ propositional variables in $\boldsymbol{y}$. For instance, in the previous example, we can use variables y_1 and y_2

in $\boldsymbol{y}$ to name actions park, move, and deliver. We can use for instance the following encoding:

$$P(\text{park}) = \neg y_1 \wedge \neg y_2 \quad P(\text{move}) = y_1 \wedge \neg y_2 \quad P(\text{deliver}) = \neg y_1 \wedge y_2.$$

Now we represent symbolically the transition function $\gamma(s)$. We will call the states in $\gamma(s)$ the *next states*. To represent next states, we need a further vector of propositional variables, say, $\boldsymbol{x}'$, of the same dimension of $\boldsymbol{x}$. Each variable $x' \in \boldsymbol{x}'$ is called a *next-state variable*. We need it because we need to represent the relation between the old and the new variables. Similarly to $P(s)$ and $P(Q)$, $P'(s)$ and $P'(Q)$ are the formulas representing state s and the set of states Q using the next state variables in $\boldsymbol{x}'$. A transition is therefore an assignment to variables in $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{x}'$

Example 12.5. Consider Example 12.4 and Example 11.1. Suppose the item to be moved is a car. The unloading operation may damage the car, and the parking area may be closed and full,[4] We have therefore some level of nondeterminism. Let x_4 and x_5 be the propositional variable for pos(car)=on_ship and pos(car)=at_harbor. The transition pos(car)=at_harbor $\in \gamma($pos(car)=on_ship, unload$)$ can be symbolically represented as[5]

$$x_4 \wedge (\neg y_1 \wedge \neg y_2) \wedge x_5',$$

which means that in the next state the car is at the harbor and may or may not be damaged. □

We define now the transition relation R corresponding to the transition function γ (this will be convenient for the definition of the symbolic representation of transition relations):

$$\forall s \in S, \forall a \in A, \forall s' \in S \; (R(s,a,s') \iff s' \in \gamma(s,a)).$$

In the rest of this section, we adopt the following notation:[6]

- Given a set of states Q, Q($\boldsymbol{x}$) is the propositional formula representing the set of states Q in the propositional variables $\boldsymbol{x}$;
- $R(\boldsymbol{x},\boldsymbol{y},\boldsymbol{x}')$ is the propositional formula in the propositional variables $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{x}'$ representing the transition relation.

We also adopt a QBF-like notation, the logic of *Quantified Boolean Formulas*, a definitional extension of propositional logic in which propositional variables can be universally and existentially quantified. According to this notation, we have:

- $\exists x Q(\boldsymbol{x})$ stands for $Q(\boldsymbol{x})[x \leftarrow \mathsf{T}] \vee Q(\boldsymbol{x})[x \leftarrow \mathsf{F}]$, where $[x \leftarrow \mathsf{T}]$ stands for the substitution of x with T in the formula;
- $\forall x Q(\boldsymbol{x})$ stands for $Q(\boldsymbol{x})[x \leftarrow \mathsf{T}] \wedge Q(\boldsymbol{x})[x \leftarrow \mathsf{F}]$.

[4]This nondeterminism models the fact that we do not know at planning time whether the parking area will be available.

[5]Here we omit the formalization of the invariant that states what does not change.

[6]Recall that a set of states is represented by a formula in state variables in $\mathbf{x}$.

Let us show how operations on sets of states and actions can be represented symbolically. Consider the set of all states s' such that from every state in Q, s' is a possible outcome of every action. The result is the set of states containing any next state s' that for any state s in Q and for any action a in A satisfies the relation $R(s, a, s')$: [7]

$$\{s' \in S \mid \forall s \in Q \text{ and } \forall a \in A.\ R(s, a, s')\}.$$

Such set can be represented symbolically with the following formula, which can be represented directly as a BDD:

$$(\exists \boldsymbol{xy}(R(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}') \wedge Q(\boldsymbol{x})))[x' \leftarrow x].$$

In this formula, the "and" operation symbolically simulates the effect of the application of any applicable action in A to any state in Q. The explicit enumeration of all the possible states and all the possible applications of actions would exponentially blow up, but symbolically we can compute all of them in a single step.

Policies are relations between states and actions, and can therefore be represented symbolically as propositional formulas in the variables $\boldsymbol{x}$ and $\boldsymbol{y}$. In the following, we write such a formula as $\pi(\boldsymbol{x}, \boldsymbol{y})$.

We are now ready to describe the planning algorithms based on symbolic model checking. In the subsequent sections, we consider an extension of the definition of planning problem where we allow for a set of initial states rather than a single initial state.

12.3.2 Planning for Safe Solutions

In Find-Safe-Solution-by-ModelChecking, Algorithm 12.9, univpol is the so-called "universal policy," that is, the set of all state-action pairs (s, a) such that a is applicable in s. Notice that starting from the universal policy may appear unfeasible in practice, because the set of all state-action pairs can be very large. We should not forget, however, that very large sets of states can be represented symbolically in a compact way. Indeed, the symbolic representation of the universal policy is:

$$\text{univpol} = \exists \boldsymbol{x}' R(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}'),$$

which also represents the applicability relation of an action in a state.

Find-Safe-Solution-by-ModelChecking calls the SafePlan routine that refines the universal policy by iteratively eliminating pairs of states and corresponding actions. This is done in two steps. First, line (*i*) removes from π' every state-action pair (s, a) for which $\gamma(s, a)$ includes a nongoal state s' that has no applicable action in π'. Next, line (*ii*) removes from π' every state-action pair (s, a) for which π' contains no path from s to the goal. This second step is performed by the routine PruneUnconnected (see Algorithm 12.10). PruneUnconnected repeatedly applies the intersection between the current policy π and the *"preimage" policy*, that is, *preimgpol* applied to the domain of the current policy and the goal states. The preimage policy,

[7]The formula is equivalent to $\bigcup_{s \in A, a \in A} \gamma(s, a)$.

```
  Find-Safe-Solution-by-ModelChecking($\Sigma, s_0, S_g$)
    univpol $\leftarrow \{(s,a) \mid s \in S$ and $a \in$ Applicable$(s)\}$
    $\pi \leftarrow$ SafePlan(univpol, $S_g$)
    if $s_0 \in (S_g \cup Domain(\pi))$ then return $\pi$
    else return(failure)

SafePlan($\pi_0$,$S_g$)
    $\pi \leftarrow \varnothing$
    $\pi' \leftarrow \pi_0$
    while $\pi \neq \pi'$ do
      $\pi \leftarrow \pi'$
      $\pi' \leftarrow \pi' \setminus \{(s,a) \in \pi' \mid \gamma(s,a) \not\subseteq (S_g \cup Domain(\pi'))\}$   $(i)$
      $\pi' \leftarrow$ PruneUnconnected($\pi', S_g$)                                $(ii)$
    return RemoveNonProgress($\pi', S_g$)                                           $(iii)$
```

Algorithm 12.9. Planning for safe solutions by symbolic model checking.

```
PruneUnconnected($\pi$,$S_g$)
    Old$\pi \leftarrow$ fail
    New$\pi \leftarrow \varnothing$
    while Old$\pi \neq$ New$\pi$ do
      Old$\pi \leftarrow$ New$\pi$
      New$\pi \leftarrow \pi \cap preimgpol(S_g \cup Domain($New$\pi))$
    return New$\pi$
```

Algorithm 12.10. PruneUnconnected: Removing unconnected states.

given a set of states $Q \subseteq S$, returns the policy that has at least one out-coming state to the given set of states:

$$preimgpol(Q) = \{(s,a) \mid \gamma(s,a) \cap Q \neq \varnothing\}.$$

$preimgpol(Q)$ is represented symbolically as a formula in the current state variables $\boldsymbol{x}$ and the action variables $\boldsymbol{y}$:

$$preimgpol(Q) = \exists \boldsymbol{x}'(R(\boldsymbol{x},\boldsymbol{y},\boldsymbol{x}') \wedge Q(\boldsymbol{x}')).$$

The pruning of outgoing and unconnected states is repeatedly performed by the while loop in SafePlan until a fixed point is reached. Then in line (iii), SafePlan removes states and corresponding actions in the policy that do not lead toward the goal. This is done by calling the RemoveNonProgress routine (see Algorithm 12.11) that repeatedly performs the pruning in two steps. First, the preimage policy preπ that leads to the domain of the policy or to the goal state in computed ("preimage policy" step). Then the states and actions that lead to the same domain of the preimage policy

```
RemoveNonProgress($\pi$,$S_g$)
    Old$\pi$ $\leftarrow$ fail
    New$\pi$ $\leftarrow$ $\varnothing$
    while Old$\pi$ $\neq$ New$\pi$ do
        pre$\pi$ $\leftarrow$ $\pi \cap preimgpol(S_g \cup Domain(\text{New}\pi))$
        Old$\pi$ $\leftarrow$ New$\pi$
        New$\pi$ $\leftarrow$ PruneStates(pre$\pi$, $S_g \cup Domain(\text{New}\pi)$)
    return New$\pi$
```

Algorithm 12.11. RemoveNonProgress: Removing states/actions that do not lead towards the goal.

or to the goal are pruned away by the PruneStates routine (let $Q \subseteq S$):

$$\text{PruneStates}(\pi, Q) = \{(s, a) \in \pi \mid s \notin Q\}.$$

The routine PruneStates that eliminates the states and actions that lead to the same domain of a policy is computed symbolically as follows:

$$\text{PruneStates}(\pi, Q) = \pi(\boldsymbol{x}, \boldsymbol{y}) \land \neg Q(\boldsymbol{x})).$$

SafePlan thus returns the policy π that has been obtained from the universal policy by removing outgoing, unconnected and nonprogressing actions. Find-Safe-Solution-by-ModelChecking finally tests whether the set of states in the returned policy union with the goal states contains all the initial states. If this is the case, π is a safe solution; otherwise no safe solution exists.

Example 12.6. Let us consider the problem on the model described in Example 11.1, initial state s_0 where pos(car)=on_ship, and goal states S_g = {pos(car)=gate2}. The "elimination" phase of the algorithm does not remove any policy from the universal policy. Indeed, the goal state is reachable from any state in the model, and therefore there are no outgoing actions. As a consequence, function RemoveNonProgress receives in input the universal policy and refines it, taking only those actions that may lead to a progress versus the goal. The sequence π_i of policies built by function RemoveNonProgress is as follows (in the following we indicate with parking1 the state where pos(car)=parking1, etc.):

Step 0	:	$\varnothing$
Step 1	:	π_1(parking1) = deliver; π_1(transit2) = move
Step 2	:	π_2(parking1) = deliver; π_2(transit2) = move; π_2(at_harbor) = park; π_2(transit1) = move
Step 3	:	π_3(parking1) = deliver; π_3(transit2) = move; π_3(at_harbor) = park; π_3(transit1) = move; π_3(parking2) = back; π_3(transit3) = back; π_3(gate1) = back; π_3(on_ship) = unload
Step 4	:	π_3

□

```
Find-Acyclic-Solution-by-ModelChecking($\Sigma, S_0, S_g$)
    $\pi_0 \leftarrow$ failure
    $\pi \leftarrow \varnothing$
    while ($\pi_0 \neq \pi$ and $S_0 \not\subseteq (S_g \cup Domain(\pi))$) do
        strongpre$\pi \leftarrow strongpreimgpol(S_g \cup Domain(\pi))$
        $\pi_0 \leftarrow \pi$
        $\pi \leftarrow \pi \cup$ PruneStates(strongpre$\pi$, $S_g \cup Domain(\pi)$)
    if ($S_0 \subseteq (S_g \cup Domain(\pi))$)
        then return $\pi$
        else return failure
```

Algorithm 12.12. Planning for acyclic solutions by symbolic model checking

A remark is in order. Algorithm 12.9 can find either safe cyclic or safe acyclic solutions. It can be modified such that it looks for a safe acyclic solution, and only if there is no such solution does it search for a safe cyclic solution (see Exercise 12.7).

12.3.3 Planning for Safe Acyclic Solutions

Find-Acyclic-Solution-by-ModelChecking (Algorithm 12.12) performs a backward breadth-first search from the goal toward the initial states. It returns a safe acyclic solution plan π if it exists, otherwise it returns failure. The policy π is constructed iteratively by the while loop. At each iteration step, the set of states S for which a safe acyclic policy has already been found is given in input to the routine *strongpreimgpol*, which returns a policy that contains the set of pairs (s, a) such that a is applicable in s and such that a leads to states which are all in $Q \subseteq S$:

$$strongpreimgpol(Q) = \{(s, a) \mid a \in Applicable(s) \text{ and } \gamma(s, a) \subseteq Q\}.$$

The routine *strongpreimgpol*, which returns a policy that contains the set of pairs (s, a) such that a is applicable in s and such that a leads to states which are all in $Q \subseteq S$:

$$strongpreimgpol(Q) = \forall \boldsymbol{x}'(R(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}') \rightarrow Q(\boldsymbol{x}')) \;\wedge\; \exists \boldsymbol{x}' R(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}'),$$

which states that any next state must be in Q and the action represented by $\boldsymbol{y}$ must be applicable. Notice that both *preimgpol*(Q) and *strongpreimgpol*(Q) are computed in one step. Moreover, policies resulting from such computation may represent an extremely large set of state-action pairs.

The routine PruneStates that eliminates the states and actions that lead to the same domain of a policy,

$$\text{PruneStates}(\pi, Q) = \{(s, a) \in \pi \mid s \notin Q\}$$

can be represented symbolically very simply by the formula

$$\pi(\boldsymbol{x}, \boldsymbol{y}) \wedge \neg Q(\boldsymbol{x})).$$

PruneStates removes from strongpreπ the pairs (s, a) such that a solution is already known. This step is what allows finding the worst-case optimal solution.

Example 12.7. Let us consider the problem on the nondeterministic model described in Example 11.1, initial set of states $S_0 = \{\text{on\_ship}\}$, and goal states $S_g = \{\text{gate1}, \text{gate2}\}$. The sequence π_i of policies built by algorithm Find-Acyclic-Solution-by-ModelChecking is as follows:

$$\begin{array}{lll}
\pi_0 & : & \emptyset \\
\pi_1 & : & \pi_1(\text{transit3}) = \text{move}; \pi_1(\text{transit2}) = \text{move} \\
\pi_2 & : & \pi_2(\text{transit3}) = \text{move}; \pi_2(\text{transit2}) = \text{move}; \\
 & & \pi_2(\text{parking1}) = \text{deliver}; \pi_2(\text{parking2}) = \text{deliver} \\
\pi_3 & : & \pi_3(\text{transit3}) = \text{move}; \pi_3(\text{transit2}) = \text{move}; \\
 & & \pi_3(\text{parking1}) = \text{deliver}; \pi_3(\text{parking2}) = \text{deliver}; \\
 & & \pi_3(\text{transit1}) = \text{move} \\
\pi_4 & : & \pi_4(\text{transit3}) = \text{move}; \pi_4(\text{transit2}) = \text{move}; \\
 & & \pi_4(\text{parking1}) = \text{deliver}; \pi_4(\text{parking2}) = \text{deliver}; \\
 & & \pi_4(\text{transit1}) = \text{move}; \pi_4(\text{at\_harbor}) = \text{park} \\
\pi_5 & : & \pi_5(\text{transit3}) = \text{move}; \pi_5(\text{transit2}) = \text{move}; \\
 & & \pi_5(\text{parking1}) = \text{deliver}; \pi_5(\text{parking2}) = \text{deliver}; \\
 & & \pi_5(\text{transit1}) = \text{move}; \pi_5(\text{at\_harbor}) = \text{park}; \\
 & & \pi_5(\text{on\_ship}) = \text{unload} \\
\pi_6 & : & \pi_5
\end{array}$$

Notice that at the fifth iteration, PruneStates removes from π_5 all the state-action pairs that move the container back (action back) from states such that a solution is already known. For instance, $\pi_5(\text{parking2}) = \text{back}$, $\pi_5(\text{gate1}) = \text{back}$, and so on. □

12.3.4 BDD-based Representation

In the previous section, we showed how the basic building blocks of the planning algorithm can be represented through operations on propositional formulas. In this section, we show how specific data structures, Binary Decision Diagrams (BDDs) , can be used for the compact representation and effective manipulation of propositional formulas.

A BDD is a directed acyclic graph (DAG). The terminal nodes are either True or False (alternatively indicated with 0 and 1, respectively). Each nonterminal node is associated with a boolean variable and with two BDDs, which are called the left and right branches. Figure 12.2 *(a)* shows a BDD for the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$.

Given a BDD, the value corresponding to a given truth assignment to the variables is determined by traversing the graph from the root to the leaves, following each branch indicated by the value assigned to the variables. A path from the root to a leaf can visit nodes associated with a subset of all the variables of the BDD. The reached leaf node is labeled with the resulting truth value. If v is a BDD, its size $|v|$ is the number of its nodes.[8] If n is a node, we will use $var(n)$ to denote the variable indexing node n. BDDs are a canonical representation of boolean formulas if

- there is a total order $<$ over the set of variables used to label nodes, such that for any node n and correspondent nonterminal child m, their variables must be ordered, $var(n) < var(m)$, and
- the BDD contains no subgraphs that are isomorphic to the BDD itself.

The choice of variable ordering may have a dramatic impact on the dimension of a BDD. For example, Figure 12.2 depicts two BDDs for the same formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$ obtained with different variable orderings.[9]

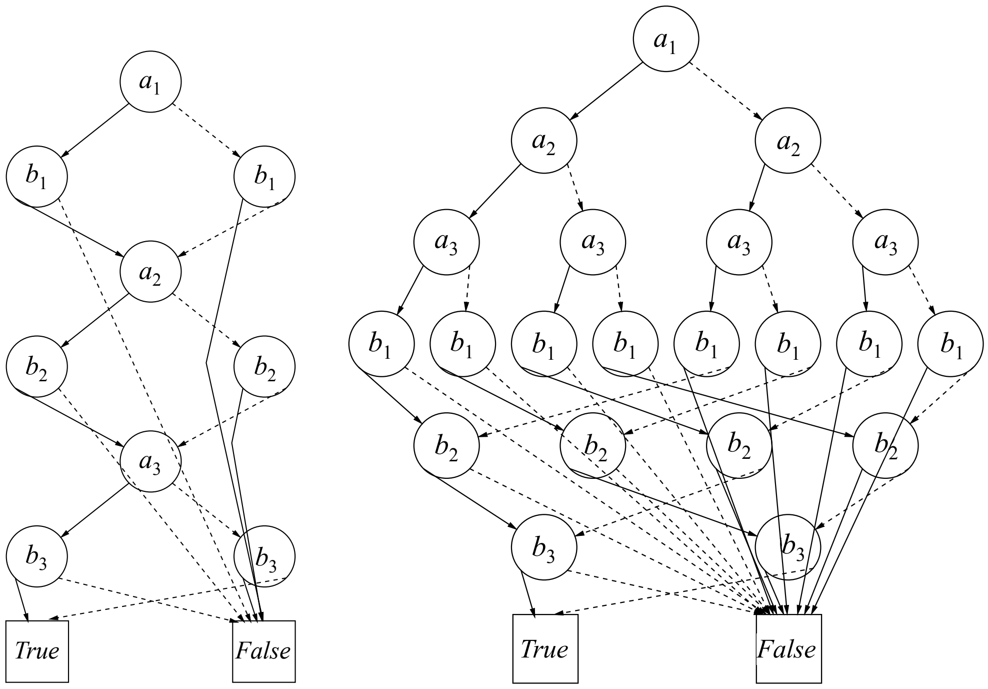


Figure 12.2. Two BDDs for the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$.

BDDs can be used to compute the results of applying the usual boolean operators. Given a BDD that represents a formula, it is possible to transform it to obtain the BDD representing the negation of the formula. Given two BDDs representing two formulas, it is possible to combine them to obtain the BDD representing the conjunction or the disjunction of the two formulas. Substitution and quantification on boolean formulas can also be performed as BDD transformations.

[8]Notice that the size can be exponential in the number of variables. In the worst case, BDDs can be very large. We do not search through the nodes of a BDD, however, but rather represent compactly (possibly very large) sets of states and work on such a representation of sets of states.

[9]A state variable representation can lead to a variable ordering in which closely related propositions are grouped together, which is critical to good performance of BDD exploration.

Comparison among different planning approaches The main advantage of determinization techniques with respect to other approaches is the possibility of exploiting fast algorithms for finding solutions that are not guaranteed to achieve the goal but just may lead to the goal, that is, unsafe solutions. Indeed, finding an unsafe solution in Σ can be done by finding a sequential plan in Σ_d. Then the sequence of actions can be easily transformed into a policy. Fast classical planners can then be used to find efficiently a solution which is unsafe for the nondeterministic model. Determinization techniques tend to work effectively when nondeterminism is limited and localized, whereas their performances can decrease when nondeterminism is high (many possible different outcomes of several actions) and in the case nondeterminism cannot be easily reconducted to exceptions of the nominal case. For these reasons, several techniques have been proposed to improve the performances when there is a high level of nondeterminisms, from conjunctive abstraction (a technique to compress states in a similar way to symbolic model checking) to techniques that exploit state relevance (see Section 12.6). With such improvements, determinization techniques have been proven to be competitive with, and in certain cases to outperform, both techniques based on And/Or search and techniques based on symbolic model checking. Finally, determinization techniques have mainly focused until now on safe cyclic planning and extensions to safe acyclic planning.

The basic idea underlying symbolic model checking techniques is to work on sets. Routines for symbolic model checking work on sets of states and on transitions from sets of states through sets of actions, rather than on single states and single state transitions. Also policies are computed and managed as sets of state-action pairs.

The symbolic model checking approach is indeed advantageous when we have a high degree of nondeterminism, that is, the set of initial states is large and several actions have many possibly different outcomes. Indeed, in these cases, dealing with a large set of initial states or a large set of outcomes of an action may have even a simpler and more compact symbolic representation than a small set. The symbolic approach may instead be outperformed by other techniques, for example, determinization techniques, when the degree of uncertainty is lower, for example, in the initial state or in the action outcomes.

12.4 Synthesis of Automata

In Section 11.2 we have defined input/output automata and how to control them with control automata (Section 11.2.1). We can specify the control automata offline once for all by hand by means of a proper programming language. It is interesting, however, to generate control automata automatically, either offline (at design time) or at run-time. Indeed, such automated synthesis, when feasible, can provide important advantages. In realistic cases, the manual specification of controllers can be difficult, time-consuming, and error prone. Moreover, in most highly dynamic domains, it is difficult if not impossible to predict all possible cases and implement a fixed controller that can deal with all of them. Synthesis of controllers at run-time can provide a way to act with deliberation taking into account the current situation and context.

In the rest of this section, we introduce a technique for the automated generation of a control automaton that interacts with (possibly several distributed) input/output automata and satisfies some desired goal, representing the objective the controller has to reach.

Consider the automaton in Figure 11.9, Section 11.2.1. We want to generate a controller Aut_c with the goal that the controlled automaton Aut reaches state s5. In order to map the synthesis problem to a planning problem as defined in Section 11.1, we must consider the fact that Aut models a domain that may be only partially observable by Aut_c. That is, at execution time, Aut_c generally has no way to find out what Aut's current state is.[10] For instance, if Aut is the input/output automaton for approaching the door in Figure 11.9, a controller Aut_c that interacts with Aut has no access to the values of Aut's internal variables, and can only deduce their values from the messages it receives. Aut_c cannot know whether or not Aut has executed the action sensedistance in Figure 11.9, that is, whether Aut is still in state s_0 (the state before executing the command) or in one of s_1 or s_2, the two states after the action has been executed. This uncertainty disappears only when one of the two outputs (far or close) is sent by Aut and received by the controller Aut_c.

We take into account this uncertainty by considering evolutions of the controlled system $Aut_c \triangleright Aut$[11] in terms of sets of states rather than states, each of them containing all the states where the controlled system could be. We have therefore to deal with sets of states rather than single states. This is a way to deal with partial observability while still making use of algorithms that work in fully observable domains.

The initial set of states is updated whenever Aut performs an observable input or output transition. If $B \subseteq S$ is the current set of states and an action $io \in I \cup O$ is observed, then the new set $B' = evolve(B, io)$ is defined as follows: $s \in evolve(B, io)$ if and only if, there is some state s' reachable from B by performing a sequence of commands, such that $s \in \gamma(s', io)$. That is, in defining $evolve(B, io)$, we first consider every evolution of states in B by internal transitions, and then, from every state reachable in this way, their evolution caused by io.

Under the assumption that the execution of actions terminates, that is, that actions cannot be trapped in loops, we can define an *Abstracted System* whose states are sets of states of the automaton and whose evolutions are over sets of states.

Definition 12.8. (Abstracted System) Let $Aut = (S, S^0, I, O, A, \gamma)$ be an automaton. The corresponding abstracted system is $\Sigma_B = (S_B, S_B^0, I, O, \gamma_B)$, where:

- S_B are the sets of states of Aut reachable from the set of possible initial states S^0,
- $S_B^0 = \{S^0\}$,
- if $evolve(B, a) = B' \neq \emptyset$ for some $a \in I \cup O$, then $B' \in \gamma_B(B, a)$. □

An abstracted system is an input/output automaton with a single initial state and no internal transitions. To define a synthesis problem in terms of a planning

[10] There might be applications in which the controller Aut_c might have access to the state of the controlled automaton Aut. However, in general, the advantage of a representation based on input/output automata is to hide or abstract away the details of the internal operations.

[11] Aut can be the parallel product of distributed automata $Aut_1 \| \ldots \| Aut_n$, see Section 11.2.2 and specifically Definition 11.16

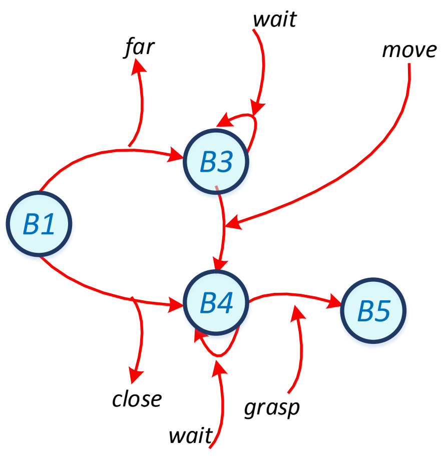


Figure 12.3. Abstracted system for the I/O automaton in Figure 11.9.

problem in nondeterministic state transitions, we need to transform an automaton $Aut = (S, S^0, I, O, A, \gamma)$ into a nondeterministic state transition system Σ. To do this transformation, we first transform the automaton into its corresponding abstracted system. This is necessary to handle partial observability and apply planning algorithms for fully observable nondeterministic models.

Example 12.9. In Figure 12.3, we show the abstracted system for the controlled automaton in Figure 11.9. States s_0, s_1, and s_2 generate the set of states $B_1 = \{s_0, s_1, s_2\}$ because there is no way for the controller to distinguish them until it receives either input far or input close by the controlled automaton. We have $B_3 = \{s_3\}$, $B_4 = \{s_4\}$, and $B_5 = \{s_5\}$. □

Given the generated abstracted system, output actions of the controlled automaton are those that cause nondeterminism. We therefore move output actions from transitions into the states of the domain and replace output transitions with nondeterministic internal transitions.

Example 12.10. In Figure 12.4, we show the nondeterministic state transition system for the automaton in Figure 12.3. We have moved the output far in state B_3 and the output close in state B_4, and transformed the two (deterministic) output transitions into one nondeterministic internal transition. Now we have the nondeterministic state transitions system $\Sigma = (S, A, \gamma)$, with states $S = \{B_1, B_3, B_4, B_5\}$, and actions $A = \{\text{farclose}, \text{wait}, \text{move}, \text{grasp}\}$, where farclose is nondeterministic. The policy

$$\pi(B_1) = \text{farclose}$$
$$\pi(B_3) = \text{move}$$
$$\pi(B_4) = \text{grasp}$$

is a safe acyclic solution for the planning problem $P = (\Sigma, B_1, B_5)$, where B_1 is the initial state and B_5 is the goal state. From π we can easily construct the control automaton Aut_c that controls Aut in Figure 11.9 and satisfies the reachability goal s_5. □

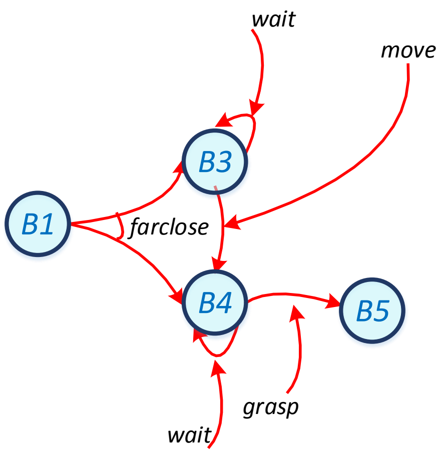


Figure 12.4. Nondeterministic model for the I/O automaton in Figure 11.9.

The synthesis problem can thus be solved by generating a nondeterministic model and by planning for a safe, possibly acyclic, policy (see Section 11.1.2) i.e., by generating a policy π that is guaranteed to reach the goal independently of the outcomes of nondeterministic transitions that are due to internal transitions and output actions of automata *Aut*.

This means that we can automate the synthesis by planning with nondetermistic models and that can find safe (acyclic) solutions, for example, planning with And/Or search (see Section 12.1), planning by symbolic model checking (see Section 12.3), and planning by determinization (see Section 12.2).

12.5 Generating Behavior Trees

Behavior Trees (BTs) have been used mainly as a high level programming language for acting, as explained in Section 11.3. BTs do not model nondeterminism explicitly but handle non-nominal executions. Hence, the generation of a BT boils down to generating a deterministic plan and augmenting the plan with appropriate conditionals. We can therefore generate BTs by planning with deterministic models, once we rely on some assumptions. Intuitively, the BT generation problem is to synthesize automatically a BT that, when executed, has the possibility to satisfy a given goal. We need to make precise what we mean by "has the possibility to satisfy the goal". Let us say that given a classical planner and planning problem, the BT generation problem seeks to synthesize a BT of root v such that BTAE(v) achieves the goal or returns failure. With this simple specification, we can devise different approaches to solve the BT generation problem. Here, we describe two simple approaches based on classical planning (Section 12.5.1), and a simple technique for interleaving BT generation and acting (Section 12.5.2).

12.5.1 BT Generation by Classical Planning

Given a classical model $\Sigma = (S, A, \gamma)$ and a classical planner, BTclassic generates a behavior tree *bt* for a problem (Σ, s, g), with s the current state at the moment of plan generation. The routine MAKEORNODE(g, π) constructs a BT *bt* whose root node is an Or node whose children are the goal g and the plan π. The plan π can be represented as an And node whose children are actions of the plan in the given planned order (see Figure 12.5):

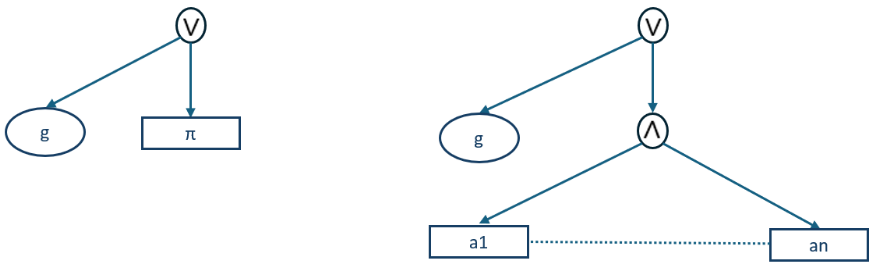


Figure 12.5. The BT generated by BTgen-by-classical-planning.

```
BTclassic($\Sigma$, $g$)
    $s$ ← observe the current state
    $\pi$ ← classical planner($\Sigma, s, g$)
    if $\pi$ = failure then return failure
    $bt$ ← MAKEORNODE($g, \pi$)
```

Algorithm 12.13. BT generation by classical planning

It is easy to show that Algorithm 12.13 solve the BT generation problem: if Algorithm 11.2 BTAE(v) does not returns failure, where v is the root node of *bt*, then BTAE(v) satisfies the goal.

The BT generated by Algorithm 12.13 is not very rich, i.e., there is no reactivity to events conditioning the applicability of actions at execution time. It is easy to generalize Algorithm 12.13 by adding in the BT each atomic precondition of each action of the generated sequential plan, and generating a plan achieving each atomic precondition, until a given desired depth in this recursive process is reached. For instance, the BT depicted in Figure 12.5 gets expanded by BTrecursive in the BT represented in Figure 12.6

However, an interesting idea for Behavior Trees is to interleave planning and acting. A different approach is to devise an algorithm that simply tries to select a primitive action a whose effects achieve a goal, if the action fails due to unsatisfied preconditions of a, the algorithm can generate an action b that is supposed to achieve the preconditions of the action a and then tries to execute the action (see Exercise 12.9).

Alternatively we may want to generate a deterministic domain Σ from the specification of Behavior Trees. The assumption is that for each node of type action we

```
BTrecursive($\Sigma, g$)
    $s \leftarrow$ observe the current state
    $\pi = \langle a_1 \dots a_n \rangle \leftarrow$ classical planner($\Sigma, s, g$)
    **if** $\pi$ = failure **then** return failure
    for $i = 1, \dots n$ expand $a_i$ with an And-node whose children are the literals
      of the preconditions of $a_i$ and finally $a_i$
    for each literal $l$ of the preconditions of $a_i$ call recursively BTrecursive($\Sigma, l$)
      until a termination condition is reached
```

Algorithm 12.14. BT generation by recursive classical planning

have a simple behavior tree of the form depicted in Figure 12.7, where, for simplicity, we assume that we have atomic preconditions and effects. In general, to represent a deterministic action with a BT, we would need a child of the And node for each literal in the preconditions, and a BT for any atomic literal in the goal. Notice that this representation is a simple rewriting of an action specification with its preconditions and effects.

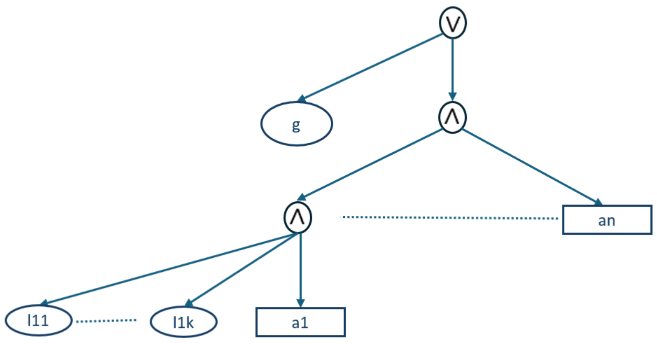
We can therefore generate a corresponding deterministic model Σ from the basic BTs representing actions (Figure 12.7). We can then use a classical planner to generate a sequential plan that we can translate back to a BT. This is essentially what does Algorithm 12.15 BTgen-from-basic BTs, a simple modification of Algorithm 12.13. $\mathcal{BT}$ is the set of basic BTs of the form represented in Figure 12.7, one for each action. BTgen-from-basic BTs organizes the basic BTs in $\mathcal{BT}$ into a plan to achieve the goal. Notice that applying BTAE recursively over the family of $\mathcal{BT}$ would not guarantee to achieve the goal.

```
BTgen-from-basic BTs($\mathcal{BT}, g$)
    $s \leftarrow$ observe the current state
    Generate $\Sigma = (S, A, \gamma)$ from $\mathcal{BT}$
    apply any of the BT generation algorithm presented in this section to ($\Sigma, g$)
    **if** BTAE over the generated BT fails **then** return failure
    return the obtained BT
```

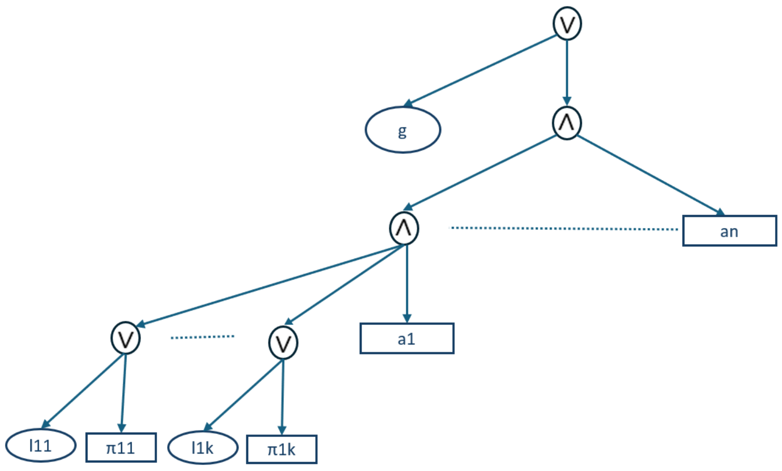
Algorithm 12.15. BT generation from basic BTs

12.5.2 BT Planning and Acting

In this section we report the simple algorithm BTplanningacting for interleaving planning and acting without the need to go through a classical specification of a deterministic model Σ but directly working on behaviour trees. Here we assume that a set of simple basic behaviour trees of the form depicted in Figure 12.7 is available. $\mathcal{BT}$ and $\mathcal{G}$ are a set of basic BTs and a set of atomic goals (a conjunction of atomic goals), respectively.



(a)



(b)

Figure 12.6. A few steps of the recursive expansion and planning by BTrecursive: (a) The first action a1 of the plan π in BT in Figure 12.5 gets expanded into an and node with all the literals l11 . . . l1k of the precondition of a1; a classical planner generates plans π11 . . . π1k to achieve the goals that are the literals f the precondition of a1: l11 . . . l1k.

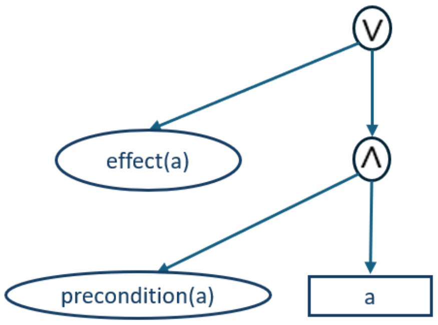


Figure 12.7. A basic BT representing an action a.

```
BTplanningacting($\mathcal{BT}, \mathcal{G}$)
1   $\mathcal{BT}' \leftarrow$ replace all $g_i \in \mathcal{G}$ with a $bt_i \in \mathcal{BT}$ having $g_i$ as effect
    foreach $bt_i \in \mathcal{BT}'$ do
        $v_i \leftarrow$ the root of $bt_i$
        while BTAE($v_i$) returns failure do
2           replace one of the atomic preconditions $l$ of $bt_i$ returning failure
             with a $bt_j \in \mathcal{BT}$ whose effect is $l$
```

Algorithm 12.16. BT planning and acting

Notice some limitations of this algorithm. It is indeed possible we have many possible bt_i that have the effects that satisfy a goal g_i, each with different preconditions (Line 1). BTplanningacting randomly selects one. This is the case also for the basic BT bt_j that replaces the atomic precondition l (Line 2). Trying to satisfy a goal or in general a condition before another may cause the impossibility to satisfy the next goal, a well-known problem in classical planning. One can provide an extension of BTplanningacting that takes this problem into account (see Section 12.6).

12.6 Discussion and Bibliographic Notes

Nondeterministic models are considered unavoidable in several areas of computer science, including computer-aided verification, model checking, control theory, and game theory.

Different techniques for planning with nondeterministic models have been developed based on different assumptions. One dimension is the degree of observability: planning with full observability (Fully Observable Nondeterminism - FOND), with partial observability (POND), and with null observability. Planning with FOND and POND is often referred as *contingent planning* or conditional planning, since the generated plans are conditional in the sense that sense the actual (set of) state(s) at execution time and provide a conditional plan depending on the actual (set of) state(s) resulting after execution. Planning with null observability generates sequential plans (no sensing is possible) and is often referred as *conformant planning*.

A second dimension is the the kind of goals: planning for reachability goals, i.e., a set of states to be reached, or for (temporally) extended goals, i.e., conditions the plan should satisfy all along the possible generated sequences of states.

A third dimension is the kind of approach and technique adopted to address the problem of planning with nondeterministic models, including different kinds of extensions of techniques for planning in deterministic models, deductive planning approaches, planning via (symbolic) model checking, techniques based on determinization, planning with belief states, planning based on temporal logic.

Probabilistic Versus Nondeterministic Approaches In Part III and Part IV, we studied two types of nondeterministic models, with and without probabilities. These models have several similarities but also differences. Let us discuss which of the two approaches to choose when faced with a practical problem where nondeterminism requires modeling and can be expressed explicitly.

Probability and cost parameters enrich the description of a domain and allow for choosing a solution according to some optimization criterion. However, estimating costs and probabilities adds a significant burden to the modeling step. There are domains in which modeling transitions with costs and probabilities is difficult in practice, for example, when not enough statistical data are available. Probabilistic approaches may also lead a modeler to hide qualitative preferences and constraints through arbitrary quantitative measures.

But there is more to it than just adding or removing parameters from one type of model to the other, as illustrated in the following example.

Example 12.11. Consider the simplistic domain in Figure 12.8 that has two policies π_a and π_b. $\pi_a(s_0) = a$; a leads to a goal with probability p in one step, or it loops back on s_0. π_b starts with action b which has a few possible outcomes, all of them lead to a goal after several steps without loops, that is, *Graph*(s_0, π_b) is acyclic, all its leaves are goal states and its paths to goals are of length $\leq k$. Both π_a and π_b are safe policies; π_a is cyclic, whereas π_b is acyclic. The value of π_a is $V_a = cost(a) \sum_{i=0,\infty}(1-p)^i = \frac{cost(a)}{p}$. The value V_b of π_b is the weighted sum of the cost of the paths of *Graph*(s_0, π_b). □

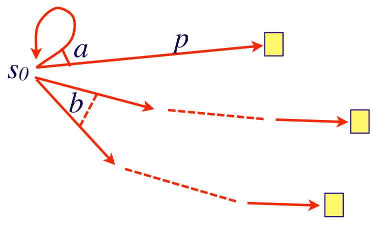


Figure 12.8. A simple domain for comparing features of probabilistic and nondeterministic models.

In this example, the probabilistic approach compares V_a to V_b and chooses the policy with the minimum expected cost. If $cost(a)/p < V_b$, then π_a is preferred to π_b, because π_b is a more expensive solution in average. However, π_b can be preferable in the *worst case* because it has bounded horizon, while for π_b it is indefinite.

The nondeterministic approach does not handle probabilities and expected costs, but it distinguishes qualitatively acyclic from cyclic solutions. There are applications in which an acyclic solution like π_b is clearly preferable whatever the values of the parameters are. This is particularly the case when these parameters are unknown or difficult to estimate. This is also the case for safety critical applications in which worst case is more meaningful than average cost.

Finally, nondeterministic approaches allow for a higher level of factorization, for example, in the case of symbolic representations.

To summarize, probabilistic approaches require parameters but are able to make fine choices on the basis of average case considerations; they allow choosing among unsafe solutions when optimizing the probability to reach a goal. Nondeterministic approaches do not need parameters and their costly estimation step; they select solutions according to a qualitative criteria

Extensions to Classical Planning. The idea of planning with nondeterministic models was first addressed in the 1980s. The first attempts to deal with nondeterminism were based on some pioneering work on conditional planning [1157, 886, 917]. This work was based on extensions to plan-space planning by extending classical planning operators to have several mutually exclusive sets of outcomes.

More recently, techniques that were originally devised for classical planning in deterministic models have been extended to deal with nondeterministic models. Planning graph techniques [146] have been extended to deal with conformant planning [1034] and some limited form of partial observability [1165]. Planning as satisfiability [591] has been extended to deal with nondeterministic models in [208, 207, 352, 423]. the work in [393] proposes a SAT encoding for planning with nondeterministic models under full observability, and an iterative SAT-based planner that effectively handles the uncertainty of FOND planning.

Different languages for representing nondeterministic actions have been investigated in [985], which defines different nondeterministic versions of PDDL, conditional STRIPS, and action languages, and the different complexity for different kinds of queries.

Deductive planning approaches. Different approaches have addressed the problem of planning with nondeterministic models in a theorem proving setting, such as techniques based on situation calculus [362] and the Golog Language [698, 777], which have also been devised for the conformant planning problem. Planning based on Quantified Boolean Formulas (QBF) has addressed the problem of conformant planning and contingent planning under partial observability [944, 947, 946]. According to this approach, a bounded planning problem, that is, a planning problem where the search is restricted to plans of maximum length n, is reformulated as a QBF formula. QBF formulas are not encoded with BDDs, instead a QBF solver is used to generate a plan. This approach can tackle several different conditional planning problems. In QBF planning, like in planning as satisfiability, it is impossible to decide the nonexistence of a solution plan.

Planinng via (symbolic) model checking. The idea of planning by using explicit state model checking techniques has been around since the work by Kabanza [564] and SimPlan, a planner that addresses the problem of planning under full observability for temporally extended goals expressed in (an extension of) Linear Temporal Logic (LTL) [325].

The idea of planning via symbolic model checking was first introduced in [234, 425]. Planning for safe acyclic solutions was first proposed in [236], and planning for safe cyclic solutions was first proposed in [235] and then revised in [271]. A full formal account and an extensive experimental evaluation of the symbolic model checking approach has been presented in [237]. The framework has been extended to deal with partial observability [125] and with extended goals [899, 901, 269]. The approach has been extnded to deal with preferences in [1002]. All the results described in the works cited have been implemented in the Model Based Planner (MBP) [124], the first planner based on the idea of planning via symbolic model checking using Binary Decision Diagrams (BDDs).

There have been various proposals along this line. The work by Jensen and Veloso [550] exploits the idea of planning via symbolic model checking as a starting point for the work on the UMOP planner. Jensen and Veloso have extended the framework to deal with contingent events in their proposal for adversarial planning [552]. They have also provided a novel algorithm for strong (safe acyclic) and strong cyclic (safe cyclic) planning which performs heuristic based guided OBDD-based search for nondeterministic models [551]. GAMER [609], the winner of the FOND track at IPC (2008), is a symbolic planner based on BDDs for safe cyclic and acyclic solutions. The planner is based on a translation of the nondeterministic planning problem into a two-player game, where actions can be selected by the planner and by the environment. Techniques for planning in nondeterministic domain models have been used to interleave reasoning about processes and ontology reasoning [903]. The Yoyo planner [660] does hierarchical planning with nondeterministic models by combining an HTN-based mechanism for constraining the search and a Binary Decision Diagram (BDD) representation for reasoning about sets of states and state transitions. In [935, 274], the synthesis of controllers from a set of components is accomplished by planning for safe acyclic solutions through model checking techniques. The work in [934] combines symbolic model-checking techniques and forward heuristic search.

Other approaches are related to model checking techniques. The work in [67] uses explicit-state model checking to embed control strategies expressed in LTL in TLPlan. The work in [434, 431, 435] presents a method where model checking with timed automata is used to verify that generated plans meet timing constraints.

Determinization Techniques. Some work on planning for cyclic safe solutions in fully observable nondeterministic domains (FOND) has focused on determinization techniques. This approach was first proposed in [659] with the NDP planner. A complete formal account and extensive experimental evaluation is presented in [25]. The planner NDP2 finds cyclic safe solutions (strong-cyclic solutions) by using a classical planner (FF). NDP2 makes use of a procedure that rewrites the original planning problem to an abstract planning problem, thus improving performances. NDP2 is compared with the MBP planner. The work in [25] shows how the performances of the two planners depend on the amount of nondeterminism in the planning domain, how the NPD2 can use effectively its abstraction mechanisms, and whether the domain contains dead ends.

A lot of interesting work has been proposed along the lines of NDP. The work in [763] proposes a planner based on the And/Or search algorithm LAO* and the pattern database heuristics to guide LAO* toward goal states. In [376], the FIP planner builds on the idea of NDP and shows how such technique can solve all of the problems presented in the international planning competition in 2008. Furthermore, FIP improves its performance by avoiding re-exploration of states that have been already encountered during the search (this idea is called *state reuse*). The work in [818], implemented in the PRP planner, devises a technique to focus on relevant aspects of states for generating safe cyclic solutions. Such technique manages to improve significantly the performance of the planner. Another study [816] extends the work to conditional effects. The work in [982] proposes a technique for conformant planning through a reduction from nondeterministic models to classical planning to find a candidate plan, and by verifying the validity of the plan with a SAT solver.

Planning based on Heuristic Search. The work in [155, 156, 159] introduced the idea of planning in belief space (i.e., the space of sets of states) using heuristic forward search. The conformant planning problem has been addressed by using SAT to reason about the effects of an action sequence and heuristic search based on FF relaxation techniques [175, 509]. The technique has been extended to deal with contingent planning in [508]. Partially observable contingent planning is further addressed in [749], a work that interleaves conformant planning with sensing actions and uses a landmark-based heuristic for selecting the next sensing action, together with a projection method that uses classical planning to solve the intermediate conformant planning problems. Another work [163] studies the complexity of belief tracking for planning with sensing both in the case of deterministic actions and uncertainty in the initial state as well as in the case of nondeterministic actions. In [887], an iterative depth-first search algorithm generates safe cyclic policies for FOND problems by exploiting the benefits of heuristic functions to make the algorithm more effective during the iterative searching process.

Planning based on Temporal Logic. Beyond the work on explicit state model checking based on Linear Time Temporal Logic (LTL) [564, 67], recent works have addressed the problem of planning in nondeterministic domains with Linear Temporal Logic and Linear Dynamic Logic on Finite Traces (LTLf and LDLf) [418], see, e.g., [417, 419, 416, 1230, 420, 153, 152, 1184].

Online approaches. Online approaches that interleave planning and acting, similar to the techniques described in Section 9.5, have been proposed also to deal with nondetermnistic models. A notable work on interleaving planning and execution in nondeterministic domains is presented in [627, 626]. These authors propose different techniques based on real-time heuristic search. Such algorithms are based on distance heuristics in the search space. The inMax Learning Real Time A* algorithm has been proposed in [626]. Indeed, algorithms devised for probabilistic planning can be used in nondeterministic domains without taking into account the probabilistic distribution. This is the case of algorithms based on sparse sampling lookahead (see Section 9.5). On one hand, these techniques allow for dealing with large planning domains that cannot be addressed by offline algorithms. On the other hand, they work on the assumption of "safely explorable" domains, that is, domains that do not contain dead ends. The work in [127] proposes a different technique based on symbolic model checking for partially observable domains, which guarantees termination in non-safely-explorable domains, still not guaranteeing to reach the goal in the unlucky case a dead end is reached. The work in [1003] focuses on fully observable domains and shows a technique that is able to interleave planning and execution in a very general and efficient way by using symbolic model checking techniques and by expressing goals that contain procedural statements (executable actions and plans) and declarative goals (formulas over state variables).

Planning as Synthesis of Controllers. The technique for the synthesis of controllers presented in Section 11.2 shares some ideas with work on the automata-based synthesis of controllers (see, e.g., [905, 906, 907, 1117, 592, 656, 655, 1115]).

In Section 12.4 we dealt with the problem of planning under partial observability by encoding in different states the different possible values of variables that cannot be observed. Work in planning under partial observability has been done in the framework of planning via symbolic model checking [128, 127, 126], real-time heuristic search [626], and heuristic search [749].

Further Recent Approaches. Some recent work addresses the FOND planning problem under explicitly provided fairness assumptions [955]. It provides a formalization of string and strong cyclic planning under a suitable fairness assumption; strong planning is represented through adversarial (or “unfair”) nondeterminism; strong-cyclic planning through fair nondeterminism under the assumption that none of the possible outcomes of a non-deterministic action can be skipped forever.

A best-first heuristic search algorithm that searches the policy-space of a FOND model has been developed in [955]. The authors generalize the concepts of optimality, admissibility, and goal-awareness to the case of FOND. In [1013], the authors address the problem of planning with unavoidable deadends, i.e., deadens that cannot be avoided by resorting to a different plan. Contingent offline and online planning is used to identify and handle unavoidable deadends. In [817], the authors propose a set of novel techniques for FOND planning to scale up to large state spaces.

The work in [780] deals with action reversibility in nondeterministic models. It defines the notions of weak and strong reversibility for non-deterministic action, i.e., whether ”all effects might be undone” (weak reversibility) and “all the effects can always be undone” (strong reversibility). It proposes techniques to determine whether an agent can always undo all possible effects of the action, or whether some of the effects can never be undone. The approach is based on the idea of encoding the concept of action reversibility into classical (deterministic) planning, to deal with weak reversibility; and FOND planning, to deal with strong reversibility.

The work in [134] proposes a ”verification via planning” approach where FOND planning is used to verify hyperproperties, i.e., properties that relate multiple paths of a computational system.

Behavior Tree Generation. Behavior Trees have been exploited with success mainly as a high level programming language for acting, as explained in Section 11.3 and discussed in Section 11.5. They have also been advocated as an approach to create deliberative agents. The BTplanningacting algorithm is an adaptation of a simple algorithm reported in [252], which calls simple BTs Postcondition-Precondition-Action BT (PPA-BT). [252] describes different extensions of the simple algorithm presented in Section 12.5.2.

Synthesis of Petri Nets. The synthesis of PNs is developed in [72]. [166, 1237] describe techniques for planning based on Petri Nets.

12.7 Exercises

12.1. Provide a definition of a "worst-case optimal" safe acyclic solution, that is, a solution that results in a path with the minimal longest distance from the goal. Rewrite algorithms for finding safe acyclic solutions (see Algorithm 12.3) by replacing the nondeterministic choice and guaranteeing that the solution is worst-case optimal.

12.2. Write deterministic algorithms for Find-Safe-Solution and Find-Acyclic-Solution, see Algorithm 12.2 and Algorithm 12.3.

12.3. Figure 12.9 is a domain model for a washing problem. To make the domain nondeterministic, suppose we assume that sometimes the start action may either succeed or fail. If it fails, it will not change the state. The run and finish actions are guaranteed to succeed. Also, say that the set of goal states S_g are all the states where $\{\text{clean(clothes)} = \text{T}, \text{clean(dishes)} = \text{T}\}$ are satisfied.

(a) Draw the state-transition system. (*Hint:* It can be rather large. To make it easier to draw, do not give names to the states, and use abbreviated names for actions.)
(b) Trace the execution of Find-Solution on this problem by drawing the And/Or search tree. The nondeterministic choices are left to the reader.
(c) Do the same for Find-Safe-Solution.
(d) Suppose A and A_d represent the set of actions in the nondeterministic model and the determinized model, respectively. Compute $|A|$ and $|A_d|$.
(e) Write down a plan π_d from the initial state to a goal state using the determinized model.

12.4. Prove that an acyclic safe solution π to the problem $P = (\Sigma, s_0, S_g)$ satisfies the condition

$$(\forall s \in \hat{\gamma}(s_0, \pi))(leaves(s, \pi) \cap S_g \neq \varnothing)) \iff leaves(s_0, \pi) \subseteq S_g.$$

12.5. Notice that Find-Acyclic-Solution-by-MinMax ignores the possibility of multiple paths to the same state. If it comes to a state s again along a different path, it does exactly the same search below s that it did before. Modify Find-Acyclic-Solution-by-MinMax such that it avoids reperforming the same search in already visited states by storing remembering the already visited states and storing the obtained solutions.

12.6. Determinization techniques rely on a transformation of nondeterministic actions into a sets of deterministic actions. Write a definition of a procedure to transform

Objects: *Items* ∪ *Machines* ∪ *Booleans* ∪ *Statuses* ∪ *Availability*

Items = {dishes, clothes},
Machines = {dw, cw} (i.e., dishwasher and clothes washer),
Booleans = {T, F},
Statuses = {idle, filling, running},
Availability = {free, using}

State variables (where $i \in$ *Items* and $m \in$ *Machines*):

clean(i) ∈ *Booleans*, status(m) ∈ *Statuses*, water ∈ *Availability*

Initial state and goal:

s_0 = {clean(dishes) = F, clean(clothes) = F, loc(dishes) = dw, loc(clothes) = cw,
status(dw) = idle, status(cw) = idle, water = free}
g = {clean(dishes) = T, clean(clothes) = T}

Action schemas (where $i \in$ *Items* and $m \in$ *Machines*):

start(m)
Pre: status(m) = idle, water = free
Eff: status(m) = filling, water = using

run(m)
Pre: status(m) = filling
Eff: status(m) = running, water = free

finish(m,i)
Pre: status(m) = running, loc(i) = m
Eff: status(m) = idle, clean(i) = T

Figure 12.9. A planning domain in which there are two devices that use water: a washing machine and a dishwasher. Because of water pressure problems, only one device can use water at a time.

a nondeterministic domain into a deterministic one. Notice that this operation is complicated by the fact that we have to take into account that in different states, the same action can lead to a set of different states. Therefore, if the set of states has exponential size with respect to the number of state variables, then this operation would generate exponentially many actions.

12.7. The algorithm for planning for safe solutions by symbolic model checking presented in this chapter (see Algorithm 12.9) can find either safe cyclic or safe acyclic solutions. Modify the algorithm such that it finds a safe acyclic solution, and only if one does not exist, does it search for a safe cyclic solution.

12.8. Consider the definition of controlled system: Definition 11.11. A control automaton A_c may be not adequate to control an automaton $A_{\|}$. Indeed, we need to guarantee that, whenever A_c sends an output, $A_{\|}$ is able to receive it as an input, and vice versa. A controller that satisfies such condition is called a deadlock-free controller. Provide a formal definition of a deadlock-free controller. Suggestion: see the paper [902] where automata are defined without commands but with τ-actions.

12.9. Write an algorithm that uses classical planning to interleave planning and acting (see Section 12.5.1)

13 Learning Nondeterministic Models

Learning for nondeterministic models can take advantage of most of the techniques developed for probabilistic models (Chapter 10). Indeed, notice that in Reinforcement Learning (RL), probabilities of action transitions are not needed, so RL techniques can be applied to nondeterministic models too. For instance, we can use the algorithms for Q-learning (Algorithm 10.1, Q-learning), parametric Q-learning (Algorithm 10.3, Parametric Q-learning), and Deep Q-learning (Algorithm 10.5, Deep Q-learning). However, these algorithms do not give explicit description models of actions.

In this section, we discuss therefore some intuitions and also some challenges of how the techniques for learning deterministic action specifications (Section 4.2) could be extended to deal with nondeterministic models. Notice however that learning lifted action schema in nondeterminism models is still an open problem. In this section, we do not present solutions, but give an idea of the research challenges and possibly directions to address such challenges in future research.

In Section 13.1, we first extend the notion of action schema to nondeterministic models. In Section 13.2, we then focus on extending the algorithms for offline action learning described in Section 4.2.1 and showing some challenges still to be addressed. The case of online action learning for nondeterministic domains might be approached in a similar way by trying to extend the algorithms presented for deterministic models (see Section 4.2.2). However an online learning approach should solve the challenge to define what is an informative state-action pair (see Algorithm 4.12) in nondeterministic models, and how to use techniques for planning in nondeterministic domains (see Algorithm 4.12 for the deterministic version) to do planning to learn actions schema. All these issues are still open problems (see also Exercise 13.3).

13.1 Nondeterministic Action Schema

In nondeterministic models, the definition of action schema (Definition 2.7, Section 2.3) must be extended to deal with different possible effects of actions. In Section 4.2, $\mathrm{pre}(\alpha(z))$ and $\mathrm{eff}(\alpha(z))$ denote the sets of preconditions and effects of action α, each of them being a set of lifted assignments of the form $x(z) = z'$, where $x(z)$ is a lifted state variable (also called structured object variable).

Action schema for nondeterministic models must be extended to take into account that actions might have different effects:

Definition 13.1. $\mathrm{Eff}(\alpha(z))$ is a nonempty set $\{\mathrm{eff}_1(\alpha(z)), \ldots, \mathrm{eff}_m(\alpha(z)\}$ of m sets of lifted assignments containing only the parameters z. □

Free pre-publication, for personal use only. To be published by Cambridge University Press.

The intuition is that $\mathrm{Eff}(\alpha(z))$ is a set of the possible different m outcomes of the application of action $\alpha(z)$. Each outcome is in its turn represented with a set of lifted assignments. If $\mathrm{Eff}(\alpha(z))$ contains just one element (just one set of lifted assignments), then α is deterministic. If $\mathrm{Eff}(\alpha(z))$ contains more than one element, then α is nondeterministic.

Notice that $\varnothing \in \mathrm{Eff}(\alpha(z)$ represents a transition that leads to the same state. If $\mathrm{Eff}(\alpha(z)) = \{\varnothing\}$, then a is an action that does nothing. This should not be confused with $\mathrm{Eff}(\alpha(z)) = \varnothing$, which is not admitted in the current formalization.

13.2 Offline Action Learning

```
Nondet-Action-Incremental-Learning($T$)
    for $a$ action name that appears in $T$ do
        $\mathrm{pre}(a(z)) \leftarrow \mathrm{U}$
        $\widehat{S} \leftarrow \varnothing$
        $i \leftarrow 1$
        $\mathrm{Eff}(a(z)) \leftarrow \mathrm{eff}_1(a(z)) \leftarrow \varnothing$
    while $T \neq \varnothing$ do
        choose $(s, a(\mathbf{c}), s') \in T$
        $\mathrm{pre}(a(z)) \leftarrow pre(a(z)) \cap s(z)$
1       if $s \in \widehat{S}$ and there exists an assignment
          $x(z) = z' \in s'(z)$ s.t. $x(z) = z'' \in \mathrm{eff}_i(a(z))$ with $z' \neq z''$ then
            $i \leftarrow i + 1$
            $\mathrm{eff}_i(a(z)) \leftarrow \mathrm{eff}_{i-1}(a(z)) \setminus \{x(z) = z' \in \mathrm{eff}_{i-1}(a(z))$ s.t. $z' \neq$
              $z''$ with $x(z) = z'' \in s'(z)\} \cup s'(z) \setminus s(z)$
            $\mathrm{Eff}(a(z)) \leftarrow \mathrm{Eff}(a(z)) \cup \mathrm{eff}_i(a(z))$
        else
2           for $k \leftarrow 1$ to $i$ do
                $\mathrm{Eff}(a(z)) \leftarrow \mathrm{Eff}(a(z)) \cap \mathrm{eff}_i(a(z))$
                $\mathrm{eff}_i(a(z)) \leftarrow \mathrm{eff}_i(a(z)) \cup s'(z) \setminus s(z)$
                $\mathrm{Eff}(a(z)) \leftarrow \mathrm{Eff}(a(z)) \cup \mathrm{eff}_i(a(z))$
        $\widehat{S} \leftarrow \widehat{S} \cup \{s\}$
        $T \leftarrow T \cap (s, a(\mathbf{c}), s')$
```

Algorithm 13.1. A simple algorithm for Incremental Nondeterministic Action Learning.

In this section, we present a first naive attempt to extend Algorithm 4.6 Action-Incremental-Learning-Simple to account for nondeterminism. This simple extension does not fully address the problem of learning correct nondeterministic action schemas, and it introduces new challenges for further research.

The underlying idea of Algorithm 13.1 Nondet-Action-Incremental-Learning is a very simple one: an action is nondeterministic when from the same state the action leads to different states. $\widehat{S}$ is the set of states visited so far. If a state s has been already encountered in a transition $(s, a(\boldsymbol{c}), s')$, and some effect $(x(\boldsymbol{z}) = z' \in s'(\boldsymbol{z}))$ is different from previous effects (Line 1), then the action is nondeterministic and we add a new set of effects to $\mathrm{Eff}(a(\boldsymbol{z}))$. If this is not the case, Algorithm 13.1 updates the current set of effects eff_i in the same way as in the deterministic case (Line 2).

However, this proposed approach has several problems. Indeed, Algorithm 13.1 detects that an action is nondeterministic when two action applications in the same state lead to different states, i.e., when T contains two transitions (s, a, s') and (s, a, s'') with $s' \neq s''$. But this is not a necessary condition. Indeed it should be possible to infer that an action is nondeterministic even when this condition does not hold, like illustrated by the following example.

Example 13.2. Consider the following simple trace, with just two transitions:

$$\begin{tikzcd} \text{x=1} \arrow[r, "\text{set(x,2)}"] & \text{x=2} \arrow[r, "\text{set(x,3)}"] & \text{x=2} \end{tikzcd} \qquad (13.1)$$

Notice that in the the first transition, the action $\mathsf{set}(x, 2)$ succeeds and the effect is $x = 2$. In the second transition, $\mathsf{set}(x, 3)$ fails and the value of x remains unchanged. We should be able to infer from the above transitions that the lifted action $\mathsf{set}(z_1, z_2)$ is nondeterministic. This should be inferred even if the action is applied to different states. The action schema that generates the above transitions should have the following preconditions and effects:

$$\begin{aligned} &\mathrm{pre}(\mathsf{set}(z_1, z_2)) = \varnothing \\ &\mathrm{Eff}(\mathsf{set}(z_1, z_2)) = \{\varnothing, \{z_1 = z_2\}\} \end{aligned}$$

However, Algorithm 13.1 does not learn the above nondeterministic effects. It erroneously learns that $\mathsf{set}(z_1, z_2)$ is a deterministic action that leads to inconsistent effects, something that is not possible at all!

$$\begin{aligned} &\mathrm{pre}(\mathsf{set}(z_1, z_2)) = \varnothing \\ &\mathrm{Eff}(\mathsf{set}(z_1, z_2)) = \{\{z_1 = z_2, \varnothing\}\} \end{aligned}$$

□

One possibility to remedy to the problem shown in Example 13.2, could be to extend Algorithm 13.1 by adding a condition to check whether the difference between the value of a state variable changes from the initial to the final state of the transition, i.e., whether $z \neq z'$, with $x(\boldsymbol{z}) = z \in s(a(\boldsymbol{z}))$ and $x(\boldsymbol{z}) = z' \in s'(a(\boldsymbol{z}))$ (see Exercise 13.1)

But this should be done with care. See the following example.

Example 13.3. In this set of transitions, in spite of the fact that the second action does not change the state, while the first does, there is no evidence of nondeterminism:

$$\begin{tikzcd} x{=}1 \arrow[r, "\text{set(x,2)}"] & x{=}2 \arrow[r, "\text{set(x,2)}"] & x{=}2 \end{tikzcd} \tag{13.2}$$

Indeed, an action schema that has only the effect of setting the value of x to 2, such as

$$\begin{aligned} \mathrm{pre}(\mathsf{set}(z_1, z_2)) &= \varnothing \\ \mathrm{Eff}(\mathsf{set}(z_1, z_2)) &= \{\{x = z_2\}\} \end{aligned}$$

can generate such a set of transitions. □

13.3 Discussion and Bibliographic Notes

While there has been a lot of work on learning action schema for deterministic models (Section 4.3) and for learning in the case of probabilistic models (Section 10.9) we are not aware of works for learning action schema in the case of nondeterministic models. Section 13.1 provides some hints about the difficulties and possible approaches to address this problem.

There is a large body of work on learning automata, from the pioneering work in [428, 44] to recent work, see, e.g., [196, 1110, 185]. Learning nondeterministic automata can be used to generate nondeterministic models $\Sigma = (S, A, \gamma)$, where γ is nondeterministic. However, notice that these approaches do not generate action schema, the resulting state transition system is specified through ground actions.

Different works deal with the problem of learning Behavior Trees, see, e.g., [253, 542] for an extensive survey, including learning BTs by genetic programming [1061, 543], generating BTs by Monte Carlo search [986], generating BTs for manipulation tasks [1135], and learning Behavior Trees based on LLMs [200].

13.4 Exercises

13.1. Modify Algorithm 13.1 to solve the problem described in Example 13.2.

13.2. Modify Algorithm 13.1 to solve the problem described in Example 13.3.

13.3. Consider the problem of extending the algorithms for online action learning in deterministic models Section 4.2.2 to deal with nondeterministic models. Describe possible approaches and open challenges.

Part V

Hierarchical Refinement Models

Let it be your constant method to look into the design of people's actions, and see what they intend to accomplish.

Marcus Aurelius, *Meditation* (Book X), circa 180

This part of the book is devoted to acting, planning and learning with operational models of actions expressed with a hierarchical task-oriented representation. We briefly discussed in Chapter 1 descriptive *versus* operational models of actions. The latter are valuable for acting. They allow for detailed descriptions of complex actions handling dynamic environments with exogenous events.

The representation relies on *hierarchical refinement methods* which describe alternative ways to handle tasks and react to events. A method can be any complex algorithm, decomposing a task into subtasks and primitive actions. Subtasks are refined recursively. Actions trigger the execution of sensory-motor procedures in closed loops that query and change the world *stochastically*.

The representation extends that of Part II with more expressive methods and nondeterminism. Because we consider actions with probabilistic outcomes, planning and learning methods rely on those seen in Part III.

In Chapter 14, we describe the representation and develop an acting system, called RAE, which uses hierarchical refinement methods that are manually specified. RAE chooses the appropriate method for the task and context at hand heuristically or with the help of a planner. In Chapter 15 we present such a planner, called UPOM, which allows for informed choices of appropriate methods for the tasks and contexts at hand. This planner uses an anytime Monte Carlo Tree Search approach, suitable for online planning. Techniques for learning heuristics to guide UPOM and for syntesizing task refinement methods are presented in Chapter 16.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

14 Acting with Hierarchical Refinement

This chapter is about a *Refinement Acting Engine* (RAE) using on a hierarchical task-oriented representation. It relies on an expressive, general-purpose language which offers rich programming control structures for online decision-making. A collection of refinement *methods* describes alternative ways to handle *tasks* and react to *events*. A method can be any complex algorithm, decomposing a task into *subtasks* and primitive *actions*. Subtasks are refined recursively. Actions trigger sensory-motor procedures that query and change the world *nondeterministically*. We assume that the methods are manually specified, and that RAE chooses the appropriate method for the task and context at hand heuristically.

RAE can handle dynamic environments and exogenous events due to other causes than the actor's own actions. It can also deal with partial and imperfect information. The value of some state variables can be unknown. A simple notation extends the range of every state variable to include a special symbol, unknown, which can be the default value of a state variable that has not been set or updated to definite value. The range of a state variable can be finite or nonfinite, discrete or continuous. Specific procedures for handling approximate values can be introduced in methods when needed.

Primitive actions may have multiple possible outcomes due to, e.g., sensing and information gathering actions, interfering exogenous events, or accidents. With RAE, the actor systematically observes which outcome actually occurs to respond accordingly. Refinement methods provide ways to condition the behavior on the observed state. However, RAE with a heuristic choices of methods, does not reason specifically on the nondeterminism of actions. It needs the planning-based look-ahead techniques of the following chapter.

Actions triggered by RAE take time to complete; multiple actions may proceed simultaneously. RAE manages an agenda of concurrent activities. In this chapter, facts are not time stamped but simply refer to the current state of the world. RAE has a limited capability for handling temporal conditions, such as temporal triggers and deadlines. Section 14.3 and Part VI discuss how to extend the representation with an explicit temporal model.

14.1 Representation

Let us define the building blocks of the task refinement methods representation.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

14.1.1 Events, Tasks and Actions

An *event* designates an observable occurrence of some type to which the actor may have to react; it corresponds to an *exogenous* change in the environment relevant for the missions and tasks of the actor and for which it has appropriate reaction models. It can be for example the activation of a typed emergency signal. It has the form event-name($args$).

A *task* is a label naming an activity to be performed. It has the form task-name($args$), where task-name designates the task considered, its arguments $args$ is an ordered list of objects and values. Tasks are hierarchically organized; *root* tasks are refined with methods into subtasks, with possible recursion.

An *action* is a primitive function with parameters that can be executed by the actor through sensory-motor commands. It has the form action-name($args$), where action-name designates the action considered, arguments $args$ is an ordered list of objects and values. An action may have *nondeterministic* effects. When the actor triggers an action a for addressing some task or event, it waits until a terminates or fails before pursuing that task or event. To follow its execution progress, when action a is triggered, there is a state variable, denoted exec-status(a) $\in$ {running, done, failed}, which expresses the fact that the execution of a is going on, has succeeded or failed. A terminated action returns a value of some type, which can be used to branch over various followup of the activity.

Example 14.1. Consider the Dock Worker Robot (DWR) domain where robot is servicing a harbor navigating in a topological map, searching for a particular container. The sets of typed objects are:

- *Robots*={r1, r2, . . . },
- *Locations*={loc1, loc2, . . . },
- *Containers*={c1, c2, . . . }.

The following state variables are kept up-to-date by the robot's execution platform:

- location(r) $\in$ *Locations*, for $r \in$ *Robots*,
- place(c) $\in$ *Robots* $\cup$ *Locations*, for $c \in$ *Containers*,
- cargo(r) $\in$ *Containers* $\cup$ {empty}, for $r \in$ *Robots*,
- view(l) $\in$ {T, F} indicates if the robot has perceived the content of location l. When view(l) = T then for every container c in l, pos(c) = l is a fact in ξ.

The robot's platform can execute the following primitive actions:

- move-to(r, l): robot r goes to adjacent location l,
- take(r, c, l): r takes container c at location l,
- put(r, c, l): r puts c in l,
- perceive(r, l): r perceives which containers are in its location l.

These actions are applicable under some conditions, for example, move-to requires the destination l to be adjacent from the current location, and take, put and perceive require r to be in l. Upon the completion of an action, the platform updates the corresponding state variables. For example, when perceive(r, l) terminates, view$(l) =$ T and pos$(c) = l$ for every container c in l.

Possible tasks for this domain can be:

- navigate(r, l): robot r goes to a reachable location l,
- fetch(r, c): r goes to where is container c and takes it,
- deliver(r, c, l): r delivers container c to a location l.

□

For the purpose of acting, we assume that we know how to perform an action, but we do not need to have an explicit model of its possible effects. These effects are observed then dealt with in methods through branches conditioned on the observed states after an action terminates or fails. In the following chapter, we'll assume given a simulator to sample through possible effects of actions in order to perform a look-ahead and allow for informed acting choices.

14.1.2 Hierarchical Refinement Methods

A refinement method is specified as a parameterized template with a name and list of arguments *method-name*$(arg_1, \ldots, arg_k)$, a *role*, a *precondition* and a *body*. The role is either a task or an event; it tells what the method is about. When the *precondition* holds in the current state, the method is *applicable* for addressing the task or event in its *role* by running a program given in the method's *body*. This program refines the task or event into a sequence of subtasks, actions, and assignments. It may use recursions and iteration loops; its sequence of steps is assumed to be finite.[1]

A ground method is obtained by the substitution of its arguments by constants that are objects and values of state variables in the domain. The instance is applicable for a task if its role matches a current task or event, and its preconditions are satisfied by the current values of the state variables. A method may have several applicable instances for a current state, task, and event. An applicable instance of a method, if executed, addresses a task or an event by refining it, in a context dependent manner, into subtasks, actions, and possibly state updates.

The body of a method is a program that refers to state variables.[2] It is also convenient to define in a body *local variables*, which are generally derived from other variables. For example, one might use stable$(o, pose) \in \{\top, \bot\}$, to designate that object o in some particular *pose* is stable; this property results from some geometric and dynamic computation. Local variables are updated by assignment statements inside methods. An assignment statement is of the form $y \leftarrow expr$, where *expr* may be either a ground value in *Range*(y), or a computational expression that returns a

[1] One way to enforce such a restriction would be as follows. For each iteration loop, one could require it to have a loop counter to terminate after a finite number of iterations. For recursions, one could use a *level mapping* (e.g., see [329, 502]) that assigns to each task t a positive integer $\ell(t)$, and require that for every method m whose task is t and every task t' that appears in the body of m, $\ell(t') < \ell(t)$. However, in most cases it is straightforward to write methods that don't necessarily satisfy this property but still don't produce infinite recursion.

[2] State variable can be viewed as global variables during the execution of the body of methods.

ground value in *Range*(y). Such an expression may include, for example, calls to specialized software packages.

The body of a method is a sequence of lines with the usual programming control structure (if-then-else, while loops, etc.), and test on the current values of state variables. A *simple* test has the form $(x \circ v)$, where x is a variable, $\circ \in \{=, \neq, <, >\}$, and v a constant in the *Range*(x). A *compound* test is a negation, conjunction, or disjunction of simple or compound tests. Tests are evaluated with respect to the current state ξ. In tests, the symbol unknown is not treated in any special way; it is just one of the state variable's possible values.

Example 14.2. Consider the task for the DWR robot in Example 14.1 to fetch a particular container c. The robot may know the place of c (i.e., this information may be in ξ), in which case the robot goes to that location to take c. Otherwise, the robot will have to look at locations whose content is unknown until it finds what it is looking for. This is expressed through the two following refinement methods:

```
m1-fetch$(r, c)$
    task: fetch$(r, c)$
     pre: place$(c) \neq$ unknown
    body: if location$(r)$ = place$(c)$ then take$(r, c,$ pos$(c))$
          else do
                navigate$(r,$ place$(c))$
                take$(r, c,$ place$(c))$
```

```
m2-fetch$(r, c)$
    task: fetch$(r, c)$
     pre: place$(c)$ = unknown
    body: if $\exists\, l \in Locations$ such that view$(l)$ = F then do
                navigate$(r, l)$
                perceive$(r, l)$
                if place$(c) = l$ then take$(r, c, l)$
                else fetch$(r, c)$
          else fail
```

□

The above example illustrates *task* refinement methods. Let us provide the robot with a method for reacting to an event.

Example 14.3. Suppose that a robot of the previous example may have to react to an emergency call by stopping its current activity and going to the location from where the emergency originates. Let us represent this with an event emergency(l, i) where l is the emergency origin location and $i \in \mathbb{N}$ is an identification number of this event. We also need an additional state variable: emergency-handling$(r) \in \{\mathrm{T}, \mathrm{F}\}$ indicates whether the robot r is engaged in handling an emergency.

```
m-emergency$(r, l, i)$
   event: emergency$(l, i)$
     pre: emergency-handling$(r)$ = F
    body: emergency-handling$(r) \leftarrow$ T
          if cargo$(r) \neq$ nil then put$(r,$ load$(r),$ location$(r))$
          move-to$(l)$
          address-emergency$(l, i)$
```

This method is applicable if robot r is not already engaged in handling an emergency. In that case, the method sets its emergency-handling state variable; it unloads whatever the robot is loaded with, if any; it triggers the action to go the emergency location, then it sets a task for addressing this emergency. Other methods are supposed to switch back emergency-handling(r) when r has finished with the task address-emergency. □

Let us now illustrate methods for the VSR domain for tasks such as opening a door. We consider a one-arm robot and assume that the door is unlocked (exercises 14.10 and 14.11 cover other cases of this domain).

Example 14.4. Let us endow the robots in the VSR domain with methods for opening doors. In addition to the state variables location, place, cargo, pose, we need to characterize the opening status of the door and the position of the robot with respect to it. The two following state variables fill that need:

- location$(r) \in$ *Locations*, for $r \in$ *Robots*,
- reachable$(r, o) \in \{$T, F$\}$: indicates that robot r is within reach of object o, here o is the door handle;
- door-status$(d) \in \{$closed, cracked, open, unknown$\}$: gives the opening status of door d, a cracked door is unlatched.

The following rigid relations are used:

- adjoining(l, d): means that location l is adjoining to door d;
- handle(d, o): o is the handle of door d;
- direction$(l, d,$ toward) or direction$(l, d,$ away): location l is on the hinge or "toward" side of door d, or on the "away" side;
- side(d, left) or side(d, right): door d turns or slides to left or to the right respectively with respect to the "toward" side of the door.
- type(d, rotates) or type(d,slide): door d rotates or slides;

The primitive actions needed to open a door are the following:

- move-close(r, o): robot r moves to a position where reachable(r, o)=T;
- move-by(r, λ): r performs a motion given by vector λ;
- grasp(r, o): robot r grasps object o;
- ungrasp(r, o): r ungrasps o;
- turn(r, o, α): r turns a grasped object o by angle $\alpha \in [-\pi, +\pi]$;
- pull(r, λ): r pulls its arm by vector λ;
- push(r, λ): r pushes its arm by λ;
- monitor-status(r, d): r focuses its perception to keep door-status updated;

- end-monitor-status(r, d): terminates the monitoring action.

We assume that actions that take absolute parameters stop when an obstacle is detected, for example, turn(r, o, α) stops when the turning reaches a limit for the rotation of o, similarly for move-by.

```
m-opendoor$(r, d, l, o)$
    task: opendoor$(r, d)$
    pre: location$(r) = l \wedge$ adjoining$(l, d) \wedge$ handle$(d, o)$
    body: while $\neg$reachable$(r, o)$ do
                move-close$(r, o)$
          monitor-status$(r, d)$
          if door-status$(d)$=closed then unlatch$(r, d)$
          throw-wide$(r, d)$
          end-monitor-status$(r, d)$
```

m-opendoor is a method for the opendoor task. It moves the robot close to the door handle, unlatches the door if it is closed, then pulls it open while monitoring its status. It has two subtasks: unlatch and throw-wide.

```
m1-unlatch$(r, d, l, o)$
    task: unlatch$(r, d)$
    pre: location$(r, l) \wedge$ direction$(l, d$, toward$) \wedge$ side$(d$, left$) \wedge$ type$(d$, rotate$)$
            $\wedge$ handle$(d, o)$
    body: grasp$(r, o)$
          turn$(r, o$, alpha1$)$
          pull$(r$, val1$)$
          if door-status$(d)$=cracked then ungrasp$(r, o)$
          else fail
m1-throw-wide$(r, d, l, o)$
    task: throw-wide$(r, d)$
    pre: location$(r, l) \wedge$ direction$(l, d$, toward$) \wedge$ side$(d$,left$) \wedge$ type$(d$, rotate$)$
            $\wedge$ handle$(d, o) \wedge$ door-status$(d)$=cracked
    body: grasp$(r, o)$
          pull$(r$, val1$)$
          move-by$(r$, val2$)$
```

The preceding two methods are for doors that open by rotating on a hinge, to the left and toward the robot. Other methods are needed for doors that rotate to the right, doors that rotate away from the robot, and sliding doors (see Exercise 14.8).

The method m1-unlatch grasps, turns, then pulls the door handle before ungrasping. The method m1-throw-wide grasps the handle, pulls, then moves backward. Here alpha1 is a positive angle corresponding to the maximum amplitude of the rotation of a door handle (e.g., about 1.5 rad), val1 is a small vector toward the robot (an amplitude of about 0.1 meter), and val2 is a larger vector backward (of about 1 meter). Other methods would survey the grasping status of what the robot is grasping, or turn the handle in the opposite direction before ungrasping it (see Exercise 14.9). □

14.1.3 Acting Domain

An acting domain is modeled as a tuple $\Sigma = (\Xi, \mathcal{T}, \mathcal{M}, A)$ where:

- Ξ is the set of world states the actor may be in;
- $\mathcal{T}$ is the set of tasks and events the actor may have to deal with;
- $\mathcal{M}$ is the set of methods for handling tasks or events in $\mathcal{T}$, and $Applicable(\xi, \tau)$ the ground methods applicable to τ in state ξ.
- A is the set of nondeterministic primitive actions the actor may perform.

Note that we do not need actions schemas and do not introduce in Σ the state transition function γ (defined by action schemas) . The dynamic of the domain is observed by performing actions in A.

Given Σ and a task or event $\tau \in \mathcal{T}$, the actor has to choose a "good" method $m \in Applicable(\xi, \tau)$ to perform τ in a current state ξ, and to follow-up the steps in the body of m, while possibly handling other tasks and events that may be requested. The actor does not require a plan, i.e., an organized set of actions or a policy: its plan for τ is given by the method m it chooses. It requires a selection procedure which designates for each task or subtask at hand the "good" method for pursuing the activity in the current context. RAE relies on a selection procedure, denoted Guide, which chooses in $Applicable(\xi, \tau)$ an appropriate method. This choice can be done heuristically or on the basis of a planner (developed in Chapter 15).

14.2 Refinement Acting Engine

The Refinement Acting Engine (RAE) reacts to tasks and events through hierarchical refinements specified by a library of methods. RAE maintains an *Agenda* consisting of a set of *refinement stacks*, one for each root task or event that needs to be addressed. A refinement stack stack is a LIFO list of tuples of the form $(\tau, m, i, tried)$ where τ is an identifier for the task or event; m is a ground method to refine τ (set to nil if no method has been chosen yet); i is a pointer to a step in the body of m, initialized to 1 (first line in the body); and *tried* is a set of ground methods already tried for τ that failed to accomplish it. The refinement stack is handled with the usual push, pop and top functions.

When RAE addresses a task τ, it must choose a ground method m for τ. This is performed by the function Guide (lines 3 in RAE, 5 in Progress and 1 in Retry). The first three arguments of Guide are the current state ξ, task τ, and stack stack; the latter contains the list *tried* such as to choose a ground method which has not been already tried. Guide is defined precisely in Chapter 15 with the help of a planner (the last two arguments d_{max} and n_{ro} are control parameters used by the planner). At this stage, let us assume that it returns a heuristic choice.

The choice of the appropriate method is with respect to the current world state ξ, which is updated not from prediction but from observation (lines 2 and 6 in RAE, 4 in Progress, and 109 in Retry). If $Applicable(\xi, \tau) \subseteq tried$, then Guide returns $\varnothing$, i.e., there is no applicable ground method for τ that has not already been tried, meaning a failure to address τ.

```
RAE
    $Agenda \leftarrow$ empty list
    while True do
1       for each new task or event $\tau$ to be addressed do
2           observe current state $\xi$
3           $m \leftarrow$ Guide$(\xi, \tau, \langle(\tau, \text{nil}, 1, \varnothing)\rangle, d_{max}, n_{ro})$
4           if $m = \varnothing$ then output($\tau$, "failed")
            else $Agenda \leftarrow Agenda \cup \{\langle(\tau, m, 1, \varnothing)\rangle\}$
5       for each stack $\in Agenda$ do
6           observe current state $\xi$
            stack $\leftarrow$ Progress(stack, $\xi$)
7           if stack $= \varnothing$ then
                $Agenda \leftarrow Agenda \setminus$ stack
                output($\tau$, "succeeded")
8           else if stack =failure then
                $Agenda \leftarrow Agenda \setminus$ stack
                output($\tau$, "failed")
```

Algorithm 14.1. RAE, a Refinement Acting Engine

The first inner loop of RAE (line 1) reads each new root task or event τ to be addressed and adds to the *Agenda* its refinement stack, initialized to $\langle(\tau, m, 1, \varnothing)\rangle$, m being the ground method returned by Guide, if there is one. The root task τ for this stack will remain at the bottom of stack until solved; the subtasks in which τ refines will be pushed onto stack along with the refinement. For each stack in *Agenda*, the second loop of RAE progresses the topmost method by one step.

To progress a refinement stack, Progress focuses on the tuple $(\tau, m, i, tried)$ at the top of stack. If the current step $m[i]$ is an action already triggered, then the execution status of this action is checked. If the action $m[i]$ is still running, this stack has to wait, but RAE goes on for other pending stacks in the *Agenda*. If $m[i]$ failed, Retry examines alternative ground methods. Otherwise the action $m[i]$ is done: RAE will proceed in the following iteration with the next step in the method m, as defined by the function Next.

Next(stack, ξ) advances within stack, as well as in the body of the topmost ground method m in stack. If i is the last step in the body of m, the current tuple is removed from stack: m has successfully addressed τ. The next tuple in stack is then considered. If τ is a root task (line 1), then Next and Progress return $\varnothing$, meaning that τ succeeded; its stack stack is removed from the *Agenda*. If i is not the last step of m, RAE proceeds to the next step j. Normally j is the next step after i, but if that step is a control instruction (e.g., an *if* or *while*) then j is the step to which the control instruction directs us (which of course may depend on the current state ξ).

Starting from line 3 in Progress, i points to the next step of m to be processed. If $m[i]$ is an assignment, the corresponding update of ξ if performed; RAE proceeds

```
Progress(stack, $\xi$)
    $(\tau, m, i, tried) \leftarrow$ top(stack)
1   if $m[i]$ is an already triggered action then          // i is the current step of m
        case exec-status $(m[i])$=
            running:  return stack
2           failed:   return Retry(stack)
            done:     return Next(stack, $\xi$)
3   else                                                    // i is the next step of m
        if $m[i]$ is an assignment step then
            update $\xi$ according to $m[i]$
            return Next(stack, $\xi$)
        if $m[i]$ is an action $a$ then
            trigger the execution of action $a$
            return stack
        if $m[i]$ is a task $\tau'$ then
4           observe current state $\xi$
5           $m' \leftarrow$ Guide$(\xi, \tau', \text{push}((\tau', nil, 1, 0), \text{stack}), d_{max}, n_{ro})$
            if $m' = \varnothing$ then return Retry(stack)
            else return push$((\tau', m', 1, \varnothing)$, stack)
```

Algorithm 14.2. Progress updates and returns stack taking into account the execution status of ongoing action or the type of the next step in method m.

```
Next(stack, $\xi$)
    repeat
        $(\tau, m, i, tried) \leftarrow$ top(stack)
        pop(stack)
1       if stack = $\langle\rangle$ then return $\varnothing$
    until $i$ is not the last step of $m$
    $j \leftarrow$ step following $i$ in $m$ depending on $\xi$
    return push$((\tau, m, j, tried)$, stack)
```

Algorithm 14.3. Next step in a ground method m for a given stack.

with the next step. If $m[i]$ is an action a, its execution is triggered; RAE will wait until a finishes to examine the Next step of m. If $m[i]$ is a task τ', a refinement with a ground method m', returned by Guide, is performed. The corresponding tuple is pushed on top of stack. If no applicable ground method is relevant for τ', then the current method m has failed to accomplish τ, so Retry is performed to find untried alternatives for m.

Retry adds the failed method m to the set of ground methods that have been tried for τ and failed. It removes the corresponding tuple from stack. It retries refining τ with another ground method m', given by Guide, which has not been already tried (line 1).

```
Retry(stack)
    $(\tau, m, i, tried) \leftarrow$ pop(stack)
    $tried \leftarrow tried \cup \{m\}$                                // $m$ failed
    observe current state $\xi$
    $m' \leftarrow$ Guide($\xi, \tau$, stack, $d_{max}, n_{ro}$)
1   if $m' \neq \varnothing$ then return push(($\tau, m', 1, tried$), stack)
2   else if stack $\neq \varnothing$ then return Retry(stack)
    else return failure
```

Algorithm 14.4. Retry examines untried alternative ground methods, if any, and returns an updated stack.

If there is no such m' and if stack is not empty (line 2), Retry calls itself recursively on the topmost stack element, which is the one that generated τ as a subtask: retrial is performed one level up in the refinement tree. If stack is empty, then τ is the root task or event: RAE failed to accomplish τ.

RAE fails either when there is no ground method applicable to the root task in the current state (line 4 of RAE), or when all applicable ground methods have been tried and failed (line 8). A method fails either when one of its actions fails (line 2 in Progress), or when all applicable ground methods for one of its subtasks have been tried and failed (line 2 in Retry).

Note that Retry is not a backtracking procedure: it does not go back to a previous *computational node* to pick up another option among the candidates that *were* applicable when that node was first reached. It finds another ground method among those that are *now* applicable for the *current* state of the world. RAE interacts with a dynamic world: it cannot rely on the set *Applicable*(ξ, τ) computed earlier, because ξ has changed, new ground methods may be applicable. However, a ground method that failed earlier may succeed later and may merit retrials. We discuss this issue in the next section.

14.3 Extending the Refinement Acting Engine

It is possible to extend RAE with a few capabilities, and possibly simplify the specification of its methods. For example it can be desirable to control the progress of tasks, e.g., to *suspend*, *resume*, or *stop* a task depending on specific conditions, including with respect to time. An application may require refining a task into *concurrent* subtasks, or controlling the order in which the agenda is progressed. It can also be desirable to have methods achieving a goal instead of performing a task. Let us discuss these extensions of RAE.

14.3.1 Goals in RAE

In some cases, an actor's objectives are more easily expressed through goals, i.e., to reach a state where some condition g holds, than through performing tasks. We can extend the task refinement method approach of RAE in order to handle goals in a restricted way. A method as a triple $(role, precondition, body)$. We can define *role* to be either a task, an event or a goal. A goal g is specified syntactically by the construct achieve(g). A few modifications to RAE are sufficient to enable it to use such methods.

This goal extension is restricted since RAE does not search for sequences of actions that can achieve a goal. Instead, it just chooses opportunistically among the methods in $\mathcal{M}$ whose roles match g. If $\mathcal{M}$ does not contain such a method, then g will not be reachable by RAE. The same actor, with the same set of actions and execution platform, might be able to reach the goal g if $\mathcal{M}$ contains a richer collection of methods. A planner is needed to overcome this limitation (see Chapter 15).

Example 14.5. Consider the task fetch of Example 14.2. Instead of refining fetch with another task, we may choose to refine it with a goal of making the position of the container c known using the following methods:

```
m-fetch($r, c$)
    task: fetch($r, c$)
     pre:
    body: achieve(pos($c$) $\neq$ unknown)
          move-to(pos($c$))
          take($r, c$)

m-find-where($r, c$)
    goal: achieve(pos($c$) $\neq$ unknown)
     pre:
    body: while there is a location $l$ such that view($l$) = F do
              move-to($l$)
              perceive($l$)
```

The last method tests its goal condition and succeeds as soon as condition is met in current state. The position of c may become known by some other means than the perceive action, e.g., if some other actor shares this information with the robot. These two methods are simpler than those in Example 14.2. □

The construct achieve(g) it is processed by RAE as a task, since it has the semantics and limitations of tasks. Its main advantage is to allow the *monitoring* of the condition g in the current state. For a method m whose role is achieve(g), RAE can check whether g holds in current ξ before starting m, while progressing it and when it finishes. If the test succeeds, then the goal is achieved, and the method stops. If the test fails when the progression finishes, then the method has failed, and Retry is performed.

In the previous example, nothing needs to be done if pos(c) is known initially; if not, the method m-find-where stops if that position becomes known at some point of the while loop. The monitoring test is easily implemented by making three modifications to the Progress procedure:

- When the previous step $m[i]$ is an action that returns failure: a Retry is performed only when g does not hold in the current ξ.

- When i is the last step of m: if g is met in the current ξ, then the top tuple is removed from the stack (success case); if not a Retry on current stack is performed.
- After i is updated with nextstep(m, i): if g is met in the current ξ, then the top tuple is removed from current stack without pursuing the refinement further.

Note that if the previous step is an action that is still running, we postpone the test until it finishes (no progress for the method in that case).

The monitoring capability can be extended for tasks by adding an extra field in methods: $(role, precondition, expected\text{-}results, body)$. The *expected-results* field is a condition to be tested in the same way as a goal.

14.3.2 Controlling the Progress of Tasks.

The need for controlling the progress of tasks is illustrated in Example 14.3. The method m-emergency is not supposed to be running in parallel with other previously started tasks. The state variable emergency-handling, when set to true, should suspend other currently running tasks.

A simple extension for controlling the progress of a task is to generalize the condition field in methods: the designer should be able to express not only preconditions, as seen earlier, but also conditions under which RAE is required to stop, suspend, or resume the progress of a task. The needed modifications are the following:

- The preconditions of a method m are checked once to define the applicable Instances$(\mathcal{M}, \tau, \xi)$; the stop and suspend conditions of m, if any, have to be tested at each call of Progress for a stack where m appears.
- This test has to be performed not only for the method m on top of the stack, but also for the methods beneath it: stopping or suspending a task means stopping or suspending the subtasks in which it is currently being refined, that is, those that are above it in the stack.
- When a task is stopped the corresponding stack is removed from the agenda; when a task is suspended, the corresponding stack remains pending with no further progress, but its resume condition is tested at each iteration of RAE to eventually pursue its progression.

Some actions may be running when a stop or suspend condition is set on: RAE has to trigger the corresponding stop or suspend orders for these actions when this is feasible.

It can be convenient to express control statements with respect to relative or absolute time. Let us assume that the value of the current time is maintained in ξ as a state variable called *now*. Alarms, watchdog timers, periodic actions, and other temporal statements can be expressed in the body of methods, for example by conditioning the progress of a task (suspend, resume, stop) with respect to values of *now*. Because the main loop of RAE progresses by just one step in the top-most methods of pending stacks, it is possible to implement a real-time control of tasks at coarse level of reactivity (see Section 14.4).

14.3.3 Retrial in RAE

As mentioned earlier, Retry is not a backtracking procedure. Since RAE interacts with a dynamic world, Retry cannot go back to a previous state. It selects a ground method among those applicable in the *current* world state, except for those that have been tried before and failed. This latter restriction may not always be necessary, since the same ground method that failed at some point may succeed later on. It can be complicated to analyze the conditions responsible for failures and ascertain whether they still hold. However, RAE can be adapted to retrial of ground methods if they are vulnerable to noisy sensing, or if the execution context is one in which they should be retried. For example, one may give the methods additional parameters that are not needed for the logic of the ground method but that characterize the context (e.g., the pose of a sensor that may have changed between trials), while bounding the number of retrials.

Retrial can be applied more easily for actions. In RAE, a ground method fails when one of its actions fails. But if an action has nondeterministic outcomes, it may be worthwhile retrying the action as assessed by its expected utility. This may be implemented simply with an ad-hoc loop on the execution-status of the actions that merit retrials. It may also rely on the computation of a near-optimal MDP policy if a probabilistic model is available.[3] Furthermore, the body of a method is a procedure in which one can specify complex retrial loops. For example, a grasp may need several of ⟨move, sense, grasp⟩ sequences before succeeding or failing.

14.3.4 Refining into Concurrent Subtasks.

RAE refines a task into sequential subtasks. It can be desirable to allow for concurrent subtasks in a refinement step. For example, a robot may have to tour a location exhaustively while concurrently performing appropriate sensing actions to correctly accomplish a perceive action.

A concurrent refinement a step in the body of a method can be expressed with a "concurrent" operator as follows:

$\{\text{concurrent: } \langle v_{1,1}, \ldots, v_{1,n}\rangle, \langle v_{2,1}, \ldots, v_{2,m}\rangle, \ldots \langle v_{k,1}, \ldots, v_{k,l}\rangle\}$

where each $\langle v_{i,1}, \ldots, v_{i,j}\rangle$ is a sequence of steps as seen so far.

The refinement of a concurrent step splits the control flow of a method into k parallel branches. The corresponding stack is split into k substacks. There is an important difference with what we saw earlier for the concurrent progression of several stacks in the agenda. The latter correspond to independent tasks that may succeed or fail independently of each others. Here, all the k substacks in which a concurrent refinement splits have to succeed before considering that concurrent refinement step as being successful.

It can also be convenient to allow RAE to order opportunistically its agenda. In Algorithm 14.1, all stacks in the agenda are progressed at each iteration. The ordering of stacks may rely on general heuristics, e.g., reacting to events first, then addressing

[3]This can be done with dummy state variables $fail_a \in \mathbb{N}$. The failure of a sets $fail_a \leftarrow fail_a + 1$; two actions are applicable to a state where $fail_a \neq 0$: a and stop-with-failure.

new tasks, before progressing on older ones. Application specific heuristics can provide precise ordering choices. More elaborate scheduling approaches may be considered to manage the agenda.

RAE's *Agenda* contains several refinement stacks, one for each top-level task, its main loop progresses these stacks concurrently. However, RAE has no built-in way to manage possible conflicts and needed synchronizations. These can be managed by the refinement methods, using semaphores. Alternatively RAE can be extended with synchronization constructs, such as those used in systems like TCA and TDL (discussed next).

14.3.5 Acting with Probabilistic Refinement Methods

An acting domain Σ can be seen globally as an MDP. We have not introduced a probabilistic transition function in the definition of Σ because RAE does not it. It explores this MDP reactively with hierarchical refinement methods. The MDP framework will useful for planning and learning in the following chapters.

Here we consider the case of a focused MDP for a particular task with probabilistic refinement methods. Instead of methods with bodies to condition RAE on the observed outcomes of actions, we may use probabilistic methods that implement a policy, *without refinements*. Let a method $m = (role, precondition, body)$ be such that its body is a policy for an SSP problem $\Sigma_m = (S_m, A_m, \gamma, \Pr, \text{cost})$, where A_m is the set of actions for m and S_m is the acting state space reduced to state variables needed in m. To address a task with a probabilistic method m we need:

(i) to find a policy π_m relevant for the states where m is applicable, and
(ii) to run π_m with procedures Run-Policy or MDP-Lookahead.

Since π_m is without refinement, problem *(ii)* is straightforward: Run-Policy or Run-Lookahead directly trigger actions $\pi_m(s)$. We can address *(i)* as a planning or a learning problem. If the probability and cost distributions can be acquired offline and are stable, and if the computation time remains compatible with acting constraints, planning algorithms of Section 9.2 can be used to compute a near optimal policy. However, these conditions are not often met. The online lookahead techniques of Section 9.5 are usually more adapted to acting with probabilistic models. This is particularly the case when a generative sampling model is available. Sampling techniques, when combined with informed heuristics V_0, are able to drive efficiently lookahead techniques. Alternatively, addressing *(i)* as a learning problem is advantageous to relieve the actor from online planning or lookahead, but also to acquire hierarchical policies with refinements (see Chapter 16).

Example 14.6. Consider Example 14.4 of opening a door. We can specify the corresponding action with a single refinement method, the model of which is partly pictured in Figure 14.1. For the sake of simplicity, the acting states are simply labeled instead of a full definition of their state variables, as described in Example 14.4. For example, s_2 corresponds to the case in which a door is closed; in s_3, it is cracked; locked and blocked are two failure cases, while open is the goal state. Furthermore, the figure does not give all applicable actions in a state, for example, there are several grasps in s_2 and s_3 (left or right hand, on “T” shaped or spherical handle) and several turns in s_4. Parameter values are also not shown. □

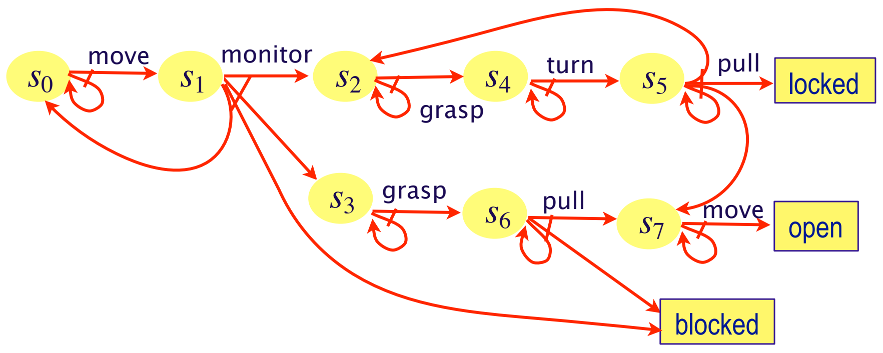


Figure 14.1. Probabilistic model for an open-door method.

Finally, note that RAE has to refine tasks but also to react to events. Probabilistic models and techniques are quite relevant when the role of a method is an event instead of a task. Probabilistic methods can be convenient for specifying reactions to unexpected events.

14.4 Discussion and Bibliographic Notes

We already discussed in Section 11.5 acting systems using automata, Petri nets and Behavior trees. Let us focus here on other representations, closer to refinement methods, that have been used for acting.

Early planning and acting systems relied on a uniform action representation with action schema. Planned actions were assument to be directly executable without refinement. This is exemplified in Planex [358], one of the first acting systems, which was coupled STRIPS. Planex assumes correct and complete state updates after each action execution; it detects failures and new opportunities for pursuing a plan. It relies on *triangle tables* to monitor the progress of a plan with respect to the goal.

The lack of robustness of this and similar systems was addressed by various approaches for specifying operational models of actions and techniques for context-dependent refinement into primitive actions. Among these, procedure-based systems are quite popular. RAP [363] is an early example. Each procedure is in charge of satisfying a particular goal, corresponding to a planned action. Deliberation chooses the appropriate package according to the current context.

RAE is inspired from the PRS system [540], a widely used procedure-based action refinement and monitoring system. In PRS one writes procedures to achieve goals or react to particular events and observations. The system commits to goals and tries alternative procedures when needed.

TCA [1024] and TDL by [1026] extend the capabilities of procedure-based systems with a wide range of synchronization constructs between commands and temporal constraints management. These and other timeline-oriented acting systems, such as RMPL of [536] are further discussed in Section 18.7.

XFRM [104] uses transformation rules to modify hand written conditional plans expressed in a representation called Reactive Plan Language [103]. It searches in plan space to improve its refinements, using simulation and probabilities of possible outcomes. It replaces the currently executed plan on the fly if it finds another one more adapted to the current situation. This approach is extended with more elaborate reactive controllers in [102].

Other procedure-based approaches have been proposed, such as IPEM by [36], EXEC by [823], or CPEF by [826]. Note also the recent ProSkill acting language [537], which maps formally to a verifiable model. Procedures in ProSkill can be proved with a formal verification system.

A few acting systems rely on logical clauses and inference mechanisms for expressing high-level specifications. Examples are the Temporal Action Logic approach of [301] for monitoring (but not action refinement) and the situation calculus approach. The latter is exemplified in GOLEX by [461], an execution system for the GOLOG planner. In GOLOG and GOLEX, the user specifies respectively planning and acting knowledge in the situation calculus representation. GOLEX provides Prolog hand-programmed “exec” clauses that explicitly define the sequence of commands a platform has to execute. It also provides monitoring primitives to check the effects of executed actions. GOLEX executes the plan produced by GOLOG, but even if the two systems rely on the same logic programming representation, they remain completely separated, limiting the interleaving of planning and acting. The PLATAS system [241] relies on GOLOG with a mapping between the PDDL langage and the Situation Calculus. The READYLOG language [353], a derivative of GOLOG, combines planning with programming. It relies on a decision-theoretic planner used by the actor when a problem needs to be solved. The actor monitors and perceives the environment through passive sensing, and acts or plans accordingly.

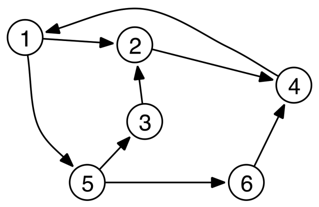
An approach based on the Business Process Execution Languages [340]), proposes to plan and compose asynchronous software services represented as state transition systems [902]. The approach produces a controller that takes into account uncertainty and the interleaving of the execution of different processes. It is extended in [188] to deal at run-time with a hierarchical representation that includes abstract actions; the problem of automated synthesis and run-time monitoring of processes is addressed in [900]. This work is further discussed in Part IV.

Many of the above systems have been used for acting as well as *monitoring*. This function fits naturally for RAE through methods for handling alarms and surveilled events. Fault detection, identification and recovery systems, e.g., in space and critical applications [531, 895], can benefit from this approach.

RAE, together with a planner and a learner, have been implemented as an open-source package with a few simulated application domains.[4]

[4]Available at https://bitbucket.org/sunandita/upom/

14.5 Exercises



$$\mathsf{road} = \{(1,2), (1,5), (2,4), (3,2),\\ (4,1), (5,3), (5,6), (6,4)\}.$$

Figure 14.2. A road network, and a rigid relation that represents it.

14.1. Suppose RAE controls a robot that travels on the network of one-way roads shown in Figure 14.2. Let the robot's current location be given by a state variable $\mathsf{loc} \in \mathit{Locations} = \{1,2,3,4,5,6\}$. For $p, q \in \mathit{Locations}$, let RAE have the following action $\mathsf{move}(p,q)$: if $\mathsf{loc} = p$ and $\mathsf{road}(p,q)$ then the robot moves from p to q (hence $\mathsf{loc} \leftarrow q$) and the action returns success. Otherwise the action returns failure.

Let $\mathsf{goto}(q)$ be the task of going to location q. Let RAE's methods for that task be:

```
m0(q)
    // already at q
    task: goto(q)
    pre: loc = q
    body:   // empty

m1(p, q)
    // move directly to q
    task: goto(q)
    pre: loc = p
         ∧ road(p, q)
    body: move(p, q)

m2(p, q, p′)
    // move to p′ then go to q
    task: goto(q)
    pre: loc = p ∧ p ≠ q
         ∧ road(p, p′)
    body: move(p, p′),
          goto(q)
```

Suppose $\mathsf{loc} = 1$ in the current state, and we give RAE the task $\mathsf{goto}(4)$. Draw the refinement tree for the shortest solution RAE can find.

14.2. Modify the m-fetch methods of Exercise 14.1 to refine a fetch into a search task of a location that has the searched container. Write an m-search methods by assuming it uses a planning function, plan-path, which computes an optimized sequence of locations with content that is not yet known; the search proceeds according to this sequence.

14.3. Complete the methods of Example 14.2 by considering that move-to is not an action but a task addressed by a method that calls a motion planner, which returns a trajectory, then controls the motion of the robot along that trajectory.

14.4. Complete the methods of Example 14.2 by considering that perceive is not an action but a task that requires calling a perception planner that returns a sequence of observation poses. Define two methods: (i) for a complete survey of a location where perceive goes through the entire sequence of observation poses and (ii) for a focus perception that stops when the searched object is detected.

14.5. Analyze how the methods in Exercises 14.2, 14.3, and 14.4 embed planning capabilities in refinement methods at the acting level.

14.6. Combine the two scenarios of Examples 14.2 and 14.3: while the robot is searching for a container, it has to react to an emergency. What needs to be done to ensure that the robot returns to its search when the task address-emergency finishes (see Section 14.3)?

14.7. In the body of the method m-opendoor of Example 14.4, why is the first word "while" rather than "if"?

14.8. Complete the methods of Example 14.4 for refining the tasks unlatch(r, d) and throw-wide(r, d) when the door turns to the right, when the door opens away from the robot, and when the door slides.

14.9. Complete the methods of Example 14.4 with appropriate steps to survey the grasping status of whatever the robot is grasping and to turn the handle in the opposite direction before ungrasping it.

14.10. Extend Example 14.4 for a robot with two arms: the robot uses its left (or right) arm if the door turns or slides to the left (or right, respectively). Add a method to move an object from one of the robot's hands to the other that can be used if the hand holding the object is needed for the opening the door.

14.11. Extend Example 14.4 for the case in which the door might be locked with an RFID lock system and the robot's RFID chip is attached to its left arm.

14.12. Redefine the pseudocode of RAE, Progress, and Retry to implement the extensions discussed in Section 14.3 for controlling the progress of a task.

14.13. Implement and test the fetch task of Example 14.2 in RAE library. [5] Integrate the results of Exercise 14.2 in your implementation; use for plan-path a simple Dijkstra graph-search algorithm. Is it possible to extend your implementation to handle the requirements stated in Exercise 14.6? Compare your implementation to the domain "exploreEnv" of RAE library where Robots and UAV move through an area and collects various data.

14.14. Here is a domain-specific acting algorithm to find near-optimal solutions for blocks world problems, where "optimal" means the smallest possible number of actions. s_0 is an initial state in which holding = nil, and g is a set of loc atoms such that:

- For each block b, if g contains an atom of the form $\text{loc}(b) = c$, then $goal(b) = c$. If there is no such atom, then $goal(b) = \text{nil}$.
- A block b is *unfinished* if $s_0(\text{loc}(b)) \neq goal(b)$ and $goal(b) \neq \text{nil}$, or if $s_0(\text{loc}(b))$ is an unfinished block. Otherwise b is *finished*.
- A block b is *clear* if $\text{top}(b) = \text{nil}$.

Here is the acting algorithm:

[5] Available at https://bitbucket.org/sunandita/upom/

```
Stack-blocks($s_0, g$)
  while there is at least one unfinished block do
    if there is an unfinished clear block $b$ such that
        $goal(b)$ = table or $goal(b)$ is a finished clear block
    then
      move $b$ to $goal(b)$
    else
      choose a clear unfinished block $b$
      move $b$ to table
```

(a) What sequence of actions will this algorithm produce for the following problem:

$$
\begin{aligned}
\mathit{Objects} &= \mathit{Blocks} \cup \{\text{hand}, \text{table}, \text{nil}\}, \mathit{Blocks} = \{\text{a}, \text{b}, \text{c}\} \\
s_0 &= \{\text{top(a)=c, top(b)=nil, top(c)=nil, holding=nil,} \\
&\qquad \text{loc(a)=table, loc(b)=table, loc(c)=a}\} \\
g &= \{\text{loc(a)=b, loc(b)=c}\}
\end{aligned}
$$

(b) Write a set of refinement methods that encode this algorithm. You may assume that there is already a function *finished*(b) that returns true if b is finished and false otherwise.

15 Hierarchical Refinement Planning

This chapter is about planning with hierarchical refinement methods. Our purpose is to guide the acting engine RAE with informed choices about the best methods for the task and context at hand. RAE uses the function Guide to choose its refinement methods. In a reactive mode, Guide performs heuristic choices. With the help of a planner, more informed and better choices can be achieved.

We consider an optimizing planner to find methods maximizing a utility function. In principle, the planner may rely on an exact dynamic programming optimization procedure. Instead, an approximation approach is more adapted to the online guidance of an actor. We describe a Monte Carlo Tree Search planner, called UPOM, parameterized for rollout depth and number of rollouts. It relies on a heuristic function for estimating the remaining of a rollout when the depth is bounded. UPOM is an anytime planner used in a receding horizon manner.

This chapter relies on chapters 8, 9 and 14. The next section presents refinement planning domains and outlines the approach. Section 15.2 proposes utility functions and an optimization procedure. The planner is developed in Section 15.3.

15.1 Refinement Planning Domains and Problems

A hierarchical refinement planning domain $\Sigma = (S, \mathcal{T}, \mathcal{M}, A)$ is defined using the same notations as an acting domain (see Section 14.1.3), with state abstraction:

- The states in S are abstractions of the acting states in Ξ. Let $\mathsf{Abstract}(\xi) \in S$ denote the abstraction of an acting state. For example a location l can be a precise metric point in Ξ and the corresponding topological label in S (see Example 14.2). Basically, $\mathsf{Abstract}(\xi)$ drops some state variables or reduces their ranges. Note that we do not have to restrict S to be finite, since we will rely on sampling.
- $\mathcal{T}$, $\mathcal{M}$, and A are respectively the sets of tasks, methods and primitive actions, as in Σ.

Actions in A are nondeterministic. They are defined with a *generative sampling model* expressed as a function Sample: $S \times A \rightarrow S \times \mathbb{R}$. Sample (s, a) returns a state s' randomly drawn among the possible outcomes of a in s, and a reward $r(s, a, s') \in \mathbb{R}$ for performing a from s to s'. Sample (s, a) may also return a token failed to account for possible failures of a.[1]

We assume (as in Definition 9.26) that calls to Sample are randomly distributed according to the probability distribution characterizing a. If $\Pr(s'|s, a)$ is available,

[1] An actor needs to know more about a failure state to react appropriately, but a simple failed token is sufficient to handle a rollout in planning.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

Sample can be implemented with random draws in this distribution. Otherwise, Sample relies on a probabilistic simulator that is assumed to reflect the true distributions.

A *planning problem* for the domain $\Sigma = (S, \mathcal{T}, \mathcal{M}, A)$ is a tuple (Σ, τ, s), for $\tau \in \mathcal{T}$ and $s \in S$. Planning for problem (Σ, τ, s) runs multiple simulations starting from s for the task τ and its subtasks with the methods in $\mathcal{M}$.

A simulation of a ground method m for task τ goes successively through the steps of m, as required by its control flow for the current context, and generates a sequence of *simulated states*: $\langle s, s_1, \ldots, s_i, \ldots \rangle$, until simulated failure, termination, or end of a rollout. It takes into account the *deterministic* pseudo-code in the body of m as well as the *nondeterministic* outcomes of its actions, as given by Sample (see Figure 15.1).

Simulation during planning does not Retry, as in RAE, but it handles a possible failure returned by Sample. Further, the planner does not observe the external world as RAE does, nor does it consider possible real world changes during a simulation. These changes are dealt with at the acting level through the main loop of RAE, which remains concurrently active during planning.

Example 15.1. Simulating the method m2-fetch(l, r) in Example 14.2 requires sampling the actions move-to(r, l) and perceive(r, l), possibly several times. It then samples action take(r, c, l) when in simulation place$(c) = l$. □

15.2 Utility Criteria and Optimal Approach

We define here the expected utility of a method from the cumulative rewards of its actions. We'll consider two different reward functions addressing different application requirements. An optimal utility will be derived, with respect to which we'll briefly discuss how may find the optimal method for a task.

15.2.1 Expected Utility of Methods

Let $U : S \times \mathcal{M} \to \mathbb{R}$ be the expected utility function of a method in a state, and $r : S \times A \times S \to \mathbb{R}$ the reward of an action in a state transition. $U(s, m)$ is computed from the cumulative rewards of the actions that m is using in a particular run. For commodity, we require $U(s, m) = 0$ when m fails. Recall that m fails if anyone of its actions fails. To easily meet this requirement, we will not cumulate the action rewards with a simple sum, but with an operator $\oplus$ that can be redefined for various reward functions, and such that $\oplus$ is associative and leads to 0 for a failed action; let $\mathbb{I}$ denote the identity element, i.e., $r(s, a, s') \oplus \mathbb{I} = \mathbb{I} \oplus r(s, a, s') = r(s, a, s')$.

In order to compute the expected utility of a method m we need to consider possible *traces* of the execution of m for a task τ. In RAE, an execution trace is conveniently represented though the evolution of stack for the task τ. In planning, we similarly use stack as defined in RAE, i.e., a LIFO list of tuples $(\tau, m, i, \mathit{tried})$.[2] For a given planning problem (τ, s), the stack corresponding to a ground method m, applicable to τ in s, is initialized as stack $= \langle (\tau, m, 1, \varnothing) \rangle$. We progress in the simulation of m

[2]We do not need for the moment to keep track of already *tried* ground methods, but we'll see in a moment the usefulness of this term.

step by step using the function Next, pushing in stack a new tuple when a step requires a refinement into a subtask.

Let top(stack) be the stack tuple $(\tau, m, i, tried)$. The utility of a particular simulation of i^{th} step of m for τ takes into account the following cases (as in Progress):

- An assignment step changes the state from s to s' but does not change the utility.
- An action a changes the state nondeterministically to s'; the utility is the cumulative reward of a with the utility of the remaining steps (operation $\oplus$). Note that the operator $\oplus$ is associative
- A refinement step does not change the state; it leads to refining τ into τ' with m'.
- The function Next moves to the following step, and to the empty stack at the end of every simulated execution.

These cases correspond to the following equation:[3]

$$U_{\mathsf{stack}}(s,m) = \begin{cases} U_{\mathsf{Next}(\mathsf{stack},s)}(s',m) & \text{if } m[i] \text{ is an assignment,} \\ r(s,a,s') \oplus U_{\mathsf{Next}(\mathsf{stack},s)}(s',m) & \\ & \text{if } m[i] \text{ is an action } a, \\ U_{\mathsf{push}((\tau',m',1,\varnothing),\mathsf{Next}(\mathsf{stack},s))}(s,m') & \\ & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if } \mathsf{stack} = \varnothing, \end{cases} \tag{15.1}$$

Let us now consider two possible action reward functions that can be used in Equation 15.1.

15.2.2 Actions Efficiency Reward

We define the efficiency reward of an action from the reciprocal of its cost. Let cost : $S \times A \times (S \cup \{\mathsf{failed}\}) \to \mathbb{R}^+$ be the cost of performing action a in state s when the outcome is s'. We assume the cost of an action a to be finite even when a fails. This is generally the case since an actor is able to figure out that an attempted action failed to limit its cost. However, a failed action a in a method m leads to the failure of m; its efficiency is simply 0. The efficiency reward function is:

$$r_e(s,a,s') = \begin{cases} 0 & \text{if } s' = \text{“failed”}, \\ 1/\mathrm{cost}(s,a,s') & \text{otherwise.} \end{cases} \tag{15.2}$$

The cumulative efficiency of two successive actions whose efficiency rewards are $r_{e1} = 1/c_1$ and $r_{e2} = 1/c_2$ is

$$r_{e1} \oplus r_{e2} = 1/(c_1 + c_2) = 1/\left(\frac{1}{r_{e1}} + \frac{1}{r_{e2}}\right) = r_{e1} \times r_{e2}/(r_{e1} + r_{e2}). \tag{15.3}$$

Note that $\oplus$ is associative; its identity is $\mathbb{I} = \infty$, corresponding to a cost of 0. Indeed, if $r_{e1} = \mathbb{I}$, then $r_{e1} \oplus r_{e2} = r_{e2}$. If either of the two actions fails than $r_{e1} \oplus r_{e2} = 0$, as required in Equation 15.1.

[3]Formally U is a function of S, $\mathcal{M}$ and the set of stacks, we keep the notation simple with a subscript.

15.2.3 Actions Success Reward

We define the success reward of an action to be 0 when it fails, and 1 if it succeeds.

$$r_s(s, a, s') = \begin{cases} 0 & \text{if } s' = \text{“failed”}, \\ 1 & \text{otherwise.} \end{cases} \tag{15.4}$$

The cumulative rewards for two actions whose success reward are r_{s1} and r_{s2} is

$$r_{s1} \oplus r_{s2} = r_{s1} \times r_{s2}. \tag{15.5}$$

Again $\oplus$ is associative; its identity is $\mathbb{I} = 1$, corresponding to success. The failure of either actions lead to a cumulative reward of zero.

15.2.4 Optimal Methods

We can replace in Equation 15.1 the function r with r_e or r_s, or any other action reward function that meet the requirements for $\oplus$. From this equation, we derive the *maximal expected utility* of m for τ by maximizing recursively over all possible refinements in m and averaging over all possible outcomes of actions, including failures:

$$U^*_{\text{stack}}(s, m) = \begin{cases} U^*_{\text{Next}(\text{stack},s)}(s', m) & \text{if } m[i] \text{ is an assignment,} \\ \sum_{s' \in \gamma(s,a)} \Pr(s'|s, a)[r(s, a, s') \oplus U^*_{\text{Next}(\text{stack},s)}(s', m)] & \\ & \text{if } m[i] \text{ is an action } a, \\ \max_{m' \in \text{Applicable}(s,\tau')} U^*_{\text{push}((\tau',m',1,\varnothing),\text{Next}(\text{stack},s))}(s, m') & \\ & \text{if } m[i] \text{ is a subtask } \tau', \\ \mathbb{I} & \text{if stack} = \varnothing. \end{cases} \tag{15.6}$$

Here $\gamma(s, a)$ is the probabilistic state transition function, including the token “failed”. It serves to define U^* even if in practice γ is not known and remains implicit in a sampling function.

The optimal ground method for a task τ in a state s for the utility U^* is:

$$m^*_{\tau,s} = \text{argmax}_{m \in \text{Applicable}(s,\tau)} U^*_{\langle(\tau,m,1,\varnothing)\rangle}(s, m) \tag{15.7}$$

If Applicable$(s, \tau) = \varnothing$ then $\max_{m \in \text{Applicable}(s,\tau)} U^* = 0$, meaning a refinement failure.

If the the probabilistic state transition function $\gamma(s, a)$ and the distribution $\Pr(s'|s, a)$ are known, it is conceivable to map Equation 15.6 into a recursive backtracking optimization algorithm, akin to dynamic programming. This algorithm would return $m^*_{\tau,s}$, as defined above. However, this knowledge of γ and the distribution Pr is not needed for RAE; it can be difficult to acquire. Moreover, the optimal planner will computationally demanding. It would not meet the needs for the online guidance of RAE. Instead, it is preferable to seek an approximately optimal method with an anytime controllable procedure. Such a procedure is developed next using a Monte Carlo Tree Search algorithm in the space of operational models.

15.3 An MCTS Planning Algorithm

UPOM is a Monte Carlo Tree Search procedure which finds an approximation $\tilde{m}$ of m^*. It performs n_{ro} rollouts, down to a depth d_{max} in the refinement tree for a task τ. It is an anytime iterative-deepening algorithm, controlled by the procedure Guide.

15.3.1 Iterative-Deepening Guidance for RAE

Recall that RAE uses Guide to choose the appropriate method. Guide receives five parameters: ξ, τ, stack_a, and two control parameters d_{max}, the maximum rollout depth, and n_{ro}, the number of rollouts. stack_a is RAE's current stack when calling Guide On a new root task τ, $\text{stack}_a = \langle(\tau, nil, 1, \varnothing)\rangle$. Guide triggers simulations starting with a stack that is a copy of stack_a, and a state s that is an abstraction of the current execution state ξ. Guide returns $\tilde{m}$, an approximately optimal ground method for τ, or $\varnothing$ if no ground method is found, that is, if there are no applicable ground methods for τ in ξ except for those already tried by RAE for this task.

Guide relies on a *method-value* function, $Q_{\text{stack}}(s, m)$, which approximates the utility $U^*_{\text{stack}}(s, m)$. This method-value function is heuristically initialized as $Q_0(s, m)$. If no planning time is available, this initialization provides a fallback policy defined as $\tilde{m} = \text{argmax}_m\, Q_0(s, m)$. Otherwise, Guide computes $Q_{\text{stack}}(s, m)$ through a succession of simulations at progressively deeper refinement levels calling UPOM for n_{ro} rollouts to evaluate the utility of a candidate ground method. $Q_{\text{stack}}(s, m)$ is maintained as a global data structure computed and updated in UPOM rollouts. The iterative-deepening loop (Line 2) is pursued until reaching the maximum depth or the search time is over. Guide returns the best method found so far: $\tilde{m} = \text{argmax}_{m \in M}\, Q_{\text{stack}}(s, m)$.

```
Guide($\xi, \tau, \text{stack}_a, d_{max}, n_{ro}$)
    $(\tau, m, i, tried) \leftarrow$ top(stack)
    $M \leftarrow$ Applicable($\xi, \tau$) $\setminus$ $tried$
    **if** $M = \varnothing$ **then** return $\varnothing$
    **if** $|M = \{m\}| = 1$ **then** return $m$
    $s \leftarrow$ Abstract($\xi$)
    stack $\leftarrow$ copy of $\text{stack}_a$; $d \leftarrow 0$
1   $\tilde{m} \leftarrow \text{argmax}_{m \in M} Q_0(s, m)$                // initialize $\tilde{m}$
2   **repeat**                                                // iterative-deepening loop
        $d \leftarrow d + 1$
3       **for** $n_{ro}$ times **do**
            UPOM ($s$, push($(\tau, nil, 1, \varnothing)$, stack), $d$)
        $\tilde{m} \leftarrow \text{argmax}_{m \in M}\, Q_{\text{stack}}(s, m)$
    **until** $d = d_{max}$ or search time is over
    return $\tilde{m}$
```

Algorithm 15.1. An iterative-deepening procedure calling UPOM to find an approximately optimal ground method.

15.3.2 A Hierarchical Refinement Planner

The procedure UPOM called by Guide performs MCTS rollouts to explore a refinement tree rooted at a planning problem (τ, s). It follows a path along the steps of chosen methods with a sampled branch from each nondeterministic node (i.e., an outcome of an action) down to depth d_{max}.

Example 15.2. Consider for example the planning problem in Figure 15.1. A rollout for τ with m can be the sequence of nodes marked in the figure as 1 (first step of m with a sample of a_1), 2 (first step of a chosen m_1 for the refinement of τ_1), ... j (subsequent steps for this refinement), $j+1$ (next step of m_1), ... n (third step of m with a sample of a_2), $n+1$ (first step of a chosen m_2 for the refinement of τ_2), etc. □

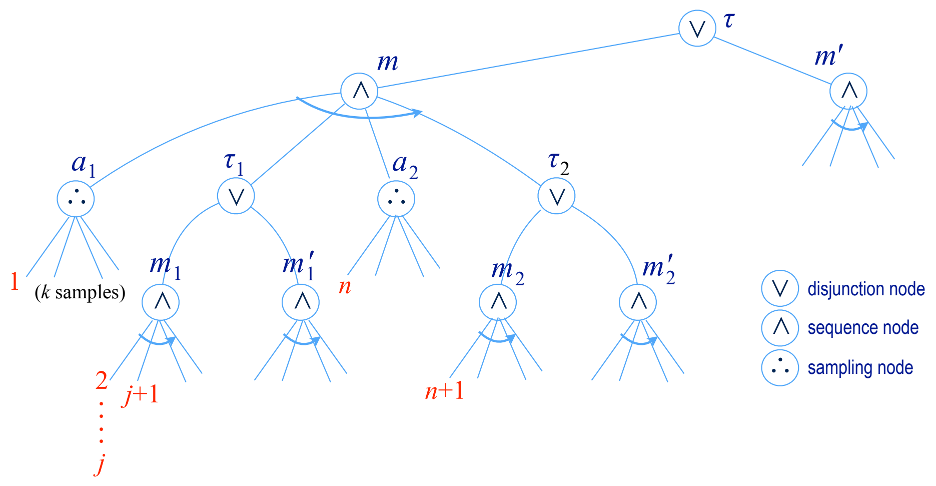


Figure 15.1. Search space for the hierarchical refinement planner. The refinement tree has three types of nodes: *disjunction* nodes for tasks over possible ground methods, *sequence* nodes for ground methods over all their steps, and *sampling* nodes for actions over their possible outcomes.

The rationale of UPOM is entailed from Equation 15.6:

- at an action node of a method m in the refinement tree, it averages the action rewards over the rollouts;
- at a task node, it chooses the refinement ground method with the highest expected utility;
- starting from the depth d given in Guide, it decreases d along recursive calls for a refinement step or an action step, but not in an assignment step;
- it takes a heuristic estimate of the utility of the remaining refinements at the tip of a rollout when reaching $d = 0$;
- it stops a rollout at a failure of an action or a refinement, and returns a value $U_{\text{Failure}} = 0$ for the corresponding method;
- it stops when the stack is empty and return $U_{\text{Success}} = \mathbb{I}$.

```
UPOM($s$, stack, $d$)
    if stack = $\langle\rangle$ then return $U_{\text{Success}}$
    $(\tau, m, i, tried) \leftarrow$ top(stack)
1   if $d = 0$ then return $Q_0(s, m)$
2   if $m \neq nil$ and $m[i]$ is an assignment then                // assignment step
        $s' \leftarrow$ state $s$ updated according to $m[i]$
        return UPOM($s'$, Next(stack, $s'$), $d$)
3   if $m \neq nil$ and $m[i]$ is an action $a$ then                 // action step
        $(s', r(s, a, s')) \leftarrow$ Sample($s, a$)
        if $s'$ = failed then return $U_{\text{Failure}}$
4       else   return $r(s, a, s') \oplus$ UPOM($s'$, Next(stack, $s'$), $d - 1$)
5   if  $m = nil$ or $m[i]$ is a task $\tau'$ then                     // refinement step
        if $m \neq nil$ then $\tau \leftarrow \tau'$
        if  $N(s, \tau)$ is not yet initialized then
            $M \leftarrow$ Applicable($s, \tau$) $\setminus$ $tried$
            if $M = 0$ then return $U_{\text{Failure}}$
            $N(s, m') \leftarrow 0$
            for  $m' \in M$ do
                $N(s, m') \leftarrow 0$ ; $Q_{\text{stack}}(s, m') \leftarrow 0$
        $Untried \leftarrow \{m' \in M | N(s, m') = 0\}$
6       if $Untried \neq \varnothing$ then
            $m_c \leftarrow$ random selection from $Untried$
7       else  $m_c \leftarrow \text{argmax}_{m \in M}\{Q_{\text{stack}}(s, m)+$
                        $C \times [\log N(s, \tau)/N(s, m)]^{1/2}\}$
8       $\lambda \leftarrow$ UPOM($s$, push(($\tau, m_c, 1, \varnothing$)), Next(stack, $s$)), $d - 1$)
9       $Q_{\text{stack}}(s, m_c) \leftarrow$
                $[N(s, m_c) \times Q_{\text{stack}}(s, m_c) + \lambda]/[1 + N(s, m_c)]$
        $N(s, m_c) \leftarrow N(s, m_c) + 1$
        $N(s, \tau) \leftarrow N(s, \tau) + 1$
        return $\lambda$
```

Algorithm 15.2. Monte Carlo tree search procedure UPOM; performs one roll-out recursively down the refinement tree of a ground method to compute an estimate of its optimal utility.

UPOM takes as arguments a simulation state s, a stack, and the rollout depth d. It performs one rollout over recursive calls for a ground method m and its refinements. On the first call of a rollout, $m = nil$, meaning that no ground method has yet been chosen. A ground method m_c is chosen among untried ground methods (Line 6). If all ground methods have been tried, m_c is chosen according to a tradeoff between exploration and exploitation (Line 7). This is the same tradeoff as in UCT for the choice of an action given in Equation 9.13. It takes into account the number $N(s, m)$ of rollouts performed from s with m, initialized to 0, and the total number $N(s, \tau)$

of rollouts for a task τ. Here $N(s,m)$ and $N(s,\tau)$ play the same role as respectively $N(s,a)$ and $N(s)$ in UCT. The constant $C > 0$ is similarly used as in UCT. It fixes the tradeoff between the exploration less sampled ground methods (high C) versus the exploitation or more promising ones (low C).

The *method-value* function $Q_{\text{stack}}(s,m)$ approximates the utility $U^*_{\text{stack}}(s,m)$. Its computation follows Equation 15.6. For an assignment step, no value update is needed, but instead a state update (Line 2). For an action step, UPOM combines the sampled action reward with the utility of the remaining part of a rollout returned by the recursion (Line 4). The action reward r (Line 4) is either r_e or r_s depending on the chosen utility function, efficiency or success ratio. For both function, $U_{\text{Success}} = \mathbb{I}$ and $U_{\text{Failure}} = 0$. For a refinement step, $Q_{\text{stack}}(s,m)$ is updated (in Line 9) by averaging over all rollouts. This update is similar to that performed by MCTS (in Line 3) or UCT (in Line 2) for $Q(s,a)$.

How UPOM departs from Equation 15.6. The pseudocode in Algorithm 15.2 departs from the specifications of Equation 15.6 in a few ways. A first one is about the use of Monte Carlo sampling instead of averaging with probability distributions, which may not be known. A second point regards the restriction of Applicable ground methods to those that have not already been tried by RAE for the same task. This is a conservative strategy, because at this point the actor has no means for distinguishing failures of tried methods that require retrials, because the world has changed, from those that don't and would fail again. We'll come back to a retrial strategy in Section 15.4.

Furthermore, Equation 15.6 does not take into account the online receding horizon usage of Guide by RAE, while UPOM does. Assume that RAE needs guidance for a root task τ (see Figure 15.1). Here the evaluations of UPOM will be similar to that of Equation 15.7 and would return, say, method m, together with its refinements m_1 for τ_1 and m_2 for τ_2. The latter choices are expressed in the method-reward function Q. Now, at acting time, when running m and reaching its refinement step of τ_1, the actor may ask again for guidance. It may rely on what was planned before, i.e., m_1. But since it as acting in dynamic world, it should better assess again its options. Here, Equation 15.6 looks down from τ_1 and does not consider the siblings of τ_1 in m. But this is not satisfactory. One should consider the utility of the methods for τ_1, as well as their impact on the remaining steps in m, i.e., on a_2 and τ_2 in this example. In other words, the actor 5164 requires the best ground method for τ_1 in the context of its current execution state, taking into account the remaining steps of the method m it is executing. This best method for τ_1 may be different from that given by Equation 15.7.

The need to keep track of the acting context with pending tasks and methods is taken care of by running Guide and UPOM with a copy of the current stack in RAE up to the root task at hand. Note however, that this does not lead to reconsider previously made choices of ground methods the actor is currently executing, e.g., in Figure 15.1, m' and other options for τ are not reassessed when seeking the best method for τ_1. Note also that UPOM does not pursue a rollout at an internal refinement node with the ground method maximizing the current utility evaluation Q, but with the best ground method according to the UCT exploration/exploitation tradeoff (Line 7).

15.3.3 Properties and Control of UPOM

Monte Carlo Tree Search approach is well adapted to planning with hierarchical refinement methods, giving an efficient and flexible anytime planner. Let us briefly discuss some of the main properties and control issues of UPOM.

Asymptotic convergence. It is possible to prove the asymptotic convergence of UPOM towards an optimal method as $n_{ro} \to \infty$. The proof assumes no depth cut-off ($d_{max} = \infty$) and static domains, i.e., domains without exogenous events. It is basically an extension of the UCT asymptotic convergence demonstration to the hierarchical refinement search space.

Control parameters. The effects of the two control parameters d_{max} and n_{ro} are not independent. This is due to the search exploration with a UCT strategy used in UPOM; it examines an untried ground method before pursuing a rollout on an already tried one. Exploration is complete if $n_{ro} > \mu$, where:

$$\mu = \textstyle\sum_{\tau_i \text{ is a subtask}} \max_s \{|\mathsf{Applicable}(s, \tau_i)|\}.$$

The sum is over all subtasks τ_i in the refinement tree, down to the refinement depth of the root task. μ increases with d_{max}. It is preferable to keep a large constant n_{ro} and increase d in the iterative-deepening loop until the max depth d_{max}.

An alternative control of Guide can be the following:

- for a given d, pursue the rollouts (Line 3 of Guide) until there are K successive exploitation rollouts, i.e., for which $Untried = \varnothing$;[4]
- pursue the iterative-deepening loop (Line 2) until no subtask is left unrefined for the K exploitation rollouts or until the search time is over.

This is an adaptive control strategy that requires only two constants K and C for the exploitation/exploration tradeoff.

Search depth. UPOM, as the Monte Carlo Tree Search algorithms of Section 9.5, runs its rollouts to some depth d. While in MDP algorithms d has only one possible meaning, here two possible definitions of d may be considered: *(i)* d is the number of task refinement steps of a rollout, or *(ii)* d is the number of refinement and action steps of a rollout. The pseudocode in Algorithm 15.2 takes the former option: d decreases at every recursive call, for an action step as well as for a task refinement step. The advantage is that the cutoff at $d = 0$ stops the current evaluation. The difficulty is that the root method, and possibly its refinements, are only partially evaluated. For example in Figure 15.1, if $j > d_{max}$, steps a_2 and τ_2 of m will never be considered; similarly for the remaining steps in m_1: a rollout might go in a deep path and never assess all the steps of evaluated ground methods. The value returned by UPOM can be arbitrarily far from U^* for small values of d. The other issue of this strategy is

[4]The probabilistic roadmap motion planning algorithms of Chapter 21 uses a similar idea to stop after K configuration samples unsuccessful for augmenting the roadmap.

that the heuristic estimate has to take into account remaining refinements lower down the cutoff point as well as remaining steps higher up in the refinement tree, i.e., what remains to be evaluated in stack.

In the alternative option, d decreases at a task refinement step only, not at an action step. The advantage is to allow each rollout to go through all the steps of every developed ground method. Furthermore, the heuristic estimate at a cutoff is focused in this case on a subtask and its applicable ground methods, whose simulation will not be started (nondeveloped ground methods). The disadvantage is that one needs an estimate of the state following the achievement of a task with a nondeveloped ground method in order to pursue the sibling steps. For example, in Figure 15.1 with $d = 1$, τ_1 will not be refined; a_2 and remaining steps of m will be based on an estimated state following the achievement of τ_1.

It is not easy to specify what might be possible default states following the achievements of a task without simulating one of its methods. If domain dependent knowledge allows such a specification, the modifications needed in UPOM to implement option *(ii)* are as follows:

- In order to be able to go back to higher levels of d when the simulation is pursued in parent ground methods after a cutoff, it is convenient to maintain d as part of the simulation stack: a fifth term d is added in every tuple of stack.
- The arguments of UPOM are modified according to the previous point.
- UPOM has to pursue in Line 1 the evaluation higher up in stack:

```
**if** $d = 0$ **then**
    return $h(\tau, m, s) \oplus$ UPOM$(g(s, \tau, m), pop($stack$), b, k)$,
```

  here $g(s, \tau, m)$ is a default state after the achievement of τ with m in s.

A mixture of the two options takes d as the number of task refinement steps in a rollout, but it stops the simulation of a method when reaching $d = 0$, without considering its remaining steps for which it takes heuristic estimates. This has the disadvantage of a partial evaluation, but it does not require estimates of following states. It requires heuristics which can be more easily defined or learned.

15.4 Discussion and Bibliographical Notes

Hierarchical refinement planning merges two techniques: MDP planning with Monte Carlo Tree Search methods (Chapter 9) and HTN planning (Chapter 5). For the latter, there is in particular a relevant link to nondeterministic HTN planning, for which [221] propose a formalization and a complexity analysis.

We already stressed the links of RAE to the goal-directed PRS system [540]. A planner called Propice-Plan [295, 538] has been developed for PRS. It performs classical planning assuming deterministic descriptive models (i.e., precond/effect) for its methods. It also performs simulations of the operational models of the methods to anticipate execution paths leading to failure. This planner has been demonstrated with PRS in mobile robotics and for the management of a blast furnace.

The UPOM planner is detailed in [872], where its performance over several domains has been analyzed. An open-source version of the code and test domains is available online.[5] This system was used in a prototype application for security monitoring and recovery from attacks on Software-Defined Networks (SDN), which has been evaluated by SDN experts successfully [873].

A significant requirement for the deployment of UPOM is the need to acquire complex domain models in order to simulate methods and actions. Once refinement methods are specified, their simulation is not an issue. Actions are given by the Sample function, which requires a simulator of the execution platform and its environment. It is not straightforward to develop a reliable simulator that reproduces the dynamics of a nondeterministic uncertain environment. In some cases, available simulation tools are very useful, e.g., physics-based simulations [149, 342, 178], robotic simulations [789, 695, 997], automated manufacturing simulations [546, 815], etc. Some of these tools are often used early on for the specification and design of an execution platform, which may simplify their later use for the development of a simulator. A fallback option, easily applicable in most cases, would be to define the procedure Sample by sampling the possible outcomes of every action from probability distributions, which are initialized by a human expert, then refined by learning and experiments. It is possible to combine detailed simulations for critical actions, for which tools might be available, and shallow simulations for the remaining actions.

The hierarchical refinement representation is very expressive, but it has nonetheless several limitation. For example, time and temporal primitives are missing. Simple temporal construct are widely used in several reactive langages, such as TCA/TDL or Petri Net based systems. Integrating similar facilities in RAE would be easy, but the extensions in UPOM will require more work. Similarly, space and movement actions are needed in robotics applications. The corresponding construct can be added in RAE, e.g., as external functions conditioning particular actions. Some of the sampling algorithms in Chapter 21 can be part of the simulations performed in UPOM.

15.5 Exercises

15.1. Define an action reward function that combines its success rate r_s with its efficiency r_e. Specify the $\oplus$ operation and $\mathbb{I}$ element for this function? Does this reward function meet the requirements of the expected utility U of Equation 15.1.

15.2. Specify the pseudocode of an algorithm akin to dynamic programming to compute the optimal method $m^*_{\tau,s}$ as defined in Equation 15.7.

15.3. Run RAE with the UPOM planner on the domain of Exercise 14.1 extended with additional methods for the tasks goto, fetch and goto, as per Exercise 14.2.

15.4. Run RAE with the UPOM planner on the domain of Exercise 14.1 extended with the methods discussed in exercises 14.3 to 14.4.

[5] https://bitbucket.org/sunandita/upom/

15.5. Extend the analysis of Exercise 14.5 about the planning capability that can be programmed in methods with those provided generically by UPOM.

15.6. Run RAE with the UPOM planner on the domain of Example 14.4 extended with the methods of exercises 14.8 to 14.11.

15.7. Assume that a domain has deterministic actions for which descriptive models are available. Revise UPOM to exploit this knowledge when all actions are deterministic. What can be done when only some actions are deterministic while others are not.

16 Learning Hierarchical Refinement Models

The hierarchical refinement approach in the previous two chapters requires *a priori domain knowledge of the methods and action models and the heuristics used by* RAE and UPOM. The topic of this chapter is to use machine learning techniques to synthesize part of this domain knowledge. Let us discuss what can be done with the learning techniques already seen:

- *Action models*. The algorithms of Chapter 4 can be extended to learn descriptive models of probabilistic action, using Sample as the oracle for online learning. One may expect these descriptive models to be faster predictors than Sample. For actions not covered by Sample, one may learn, online or offline, the models from traces. But obtaining traces may not be easy.
- *Method models*. The techniques for learning HTN methods in Chapter 7 may possibly be extended for learning refinement methods whose bodies are restricted to sequences of refinement steps and probabilistic actions. This might be of interest for planning (as HTN methods are mostly beneficial to planning), but the restriction can be a disadvantage for acting.
- *Task know-how*. The reinforcement learning algorithms of Chapter 10 can be used to synthesize policies for tasks (assuming actions have reward functions). Furthermore, we can use hierarchical RL techniques to synthesize for a task a collection of methods whose bodies are policies (see Section 14.3.5).

The details of all these options would require extensive developments. We focus instead this chapter on learning heuristics and methods using operational models with a Sample function. Section 16.1 considers learning domain-dependent heuristics to guide RAE and UPOM. It offers an illustration of the "planning to learn" idea: given methods and a Sample function, UPOM generates the near-optimal choices that are taken as targets by a deep Q-learning procedure. Section 16.2 synthesizes methods for tasks using hierarchical reinforcement techniques.

16.1 Learning to guide RAE and UPOM

Assume a hierarchical refinement domain $\Sigma = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$ is given. S may have state variables that are reals or vectors over the reals, e.g., images. As seen in Section 10.5, high-dimensional data can be handled with relevant sampling and continual learning.

Recall that RAE relies on Guide to choose its methods. If there is no time to call

Free pre-publication, for personal use only. To be published by Cambridge University Press.

UPOM, Guide returns a fallback policy $\pi(s) = \tilde{m}$, with:

$$\tilde{m} = \operatorname*{argmax}_m Q_0(s, m)$$

where Q_0 is a heuristic estimate of the method-value function. Otherwise, UPOM performs a series of progressively deeper rollouts. At the end of each rollout (that is, when $d = 0$), UPOM uses $Q_0(s, m)$ as an estimate of the utility of the remaining refinements, to compute the method-value function $Q_{\mathsf{stack}}(s, m)$ for the methods applicable to the task at hand.

Recall that the function $Q_0 : S \times \mathcal{M} \to \mathbb{R}^+$ approximates the method-value function Q for a state, a method and its corresponding task.[1] Q_0 ignores the details of pending activities in a stack, if any.[2] However, it is of direct use for acting reactively in Guide, as well as for planning with UPOM.

Consequently, learning a good Q_0 function is very beneficial. This can be done from a representative training set of state-method pairs, using a planner with simulation. Simulation is at the core of the hierarchical refinement approach, since $\mathcal{M}$ provides the simulation code for the refinements and $\mathcal{A}$ is defined with Sample through a simulator. The problem is formulated as a supervised or self-supervised regression learning problem, for which one can use neural networks and apply the techniques in Chapter 10. The next section describes the main stages for addressing this problem.

16.1.1 Training set generation

Let us call UPOM on randomly generated planning problems (τ, s) with stack $= \langle(\tau, nil, 1, \varnothing)\rangle$ and, if possible, $d = \infty$, otherwise with d very large and an arbitrary initial function Q_0. For each problem (τ, s), we record the method-value function $Q_{\mathsf{stack}}(s, m)$ computed by UPOM for all ground methods applicable in s. This gives the following training set, where m is a ground method:

$$\mathcal{D} = \{((s, m), Q_{\mathsf{stack}}(s, m)) \mid m \text{ is applicable to } (\tau, s) \in \mathit{Training}\}.$$

It is easy to restrict the training set generation to states for which the task τ has applicable methods. Note that for the same state, there may be several ground methods applicable to a task. For example, in Example 14.3 the method m-emergency(r, l, i) has as many instances as there are available robots. Knowledge about the set of states in which a task is most relevant can be very helpful for learning from a focused and appropriate training set.

We need to keep the training set $\mathcal{D}$ as representative as possible of the actual distribution of the problems that RAE/UPOM will see. This means that the randomly generated problems (τ, s) must follow such a distribution. It can be difficult to estimate such a distribution and to draw instances accordingly (especially when S is high-dimensional, e.g., for domain with visual sensing state variables). Fortunately, continual learning will allow us to compensate for a partial or biased training set $\mathcal{D}$.

[1] Recall that a method is specific to a single task.
[2] There are no pending activities for a root task, for which stack $= \langle(\tau, nil, 1, \varnothing)\rangle$.

16.1.2 Data encoding

The training pairs (s, m) will be taken as input by a learner; they need to be appropriately encoded as vectors of numeric features $\boldsymbol{\phi} = [\phi_1, \ldots, \phi_n]^\top$ for parametric neural net approximators. A popular encoding for symbolic state variables in the input is the so-called One-Hot binary encoding. For example, if a variable has n possible values $\{v_1, \ldots, v_n\}$, then each v_i is encoded as a binary vector of length n in which the i'th value is 1 and all other values are 0.

Other more efficient encodings of symbolic variables can be used that are akin to word embeddings and require a prior learning step specific to $\mathcal{D}$. Furthermore, a number of data processing techniques have been developed to improve the encoding. These include, for example, feature-selection techniques to reduce the size of the input vectors, feature normalization to reduce the variability of the data, and feature representation in domain adaptation methods. These data processing techniques are briefly discussed in Section 16.3.

16.1.3 Neural Net Training

Several domain-dependent design choices (discussed in Chapter 10) have to be made for this supervised regression learning problem. A network architecture has to be chosen, such as a feedforward net with a number of hidden layers, with the inputs being vectors $\boldsymbol{\phi}$ that encode the pairs (s, m), and the outputs being scalars for the targets $Q_{\text{stack}}(s, m)$. Nonlinear neural triggering functions and a loss function have to be chosen.

A version of the Backpropagation gradient descent algorithm, in mini-batch mode with appropriate hyperparameters, can be used for estimating the neural net parameters.

In addition to the training set $\mathcal{D}$, we need a qualification set to assess the quality of the resulting network and improve the learning as needed. This qualification set can be generated with UPOM simulations, as done for the synthesis of $\mathcal{D}$.

16.1.4 Continual Online Incremental Learning

Even with good simulators and expert knowledge to drive the sampling of training problems, the training data may not reflect an actor's specific working conditions. This is a well known and important problem in machine learning.

There are two issues for the problem considered here:

- State variables are not independent, i.e., sampling assuming independence can be biased, and tasks may naturally arise only in specific states.
- The assumption that Sample (s, a) reflects the true distribution is hard to meet precisely, leading to possible gaps between the estimated method-value function $Q_{\text{stack}}(s, m)$ and the true utility function.

These issues can be addressed by continual reinforcement learning. A procedure to do so can be the following.

Initialization:

- Rely on simulated data for the offline learning, as described above, of a neural net denoted $[Q_\theta]$.
- Use $[Q_\theta]$ to compute the estimates $Q_0(m, s)$ when needed in Guide and UPOM.

Online acting, planning and incremental learning:

- Act online with RAE and UPOM using $[Q_\theta]$, and
- Continually update the parameters of $[Q_\theta]$ using CORL algorithm.

```
CORL
    initialize network $[Q_\theta]$ and replay-memory $\mathcal{R}_M$ with simulated data
    $[\hat{Q}_{\theta^-}] \leftarrow [Q_\theta]$                                  // target network
    while True do
1       record pair $(s, m)$ when RAE calls Guide in $s$ and gets $m$
2       record resulting pair $(s', U(s, m))$ when RAE terminates with $m$
        push$((s, m, s', U(s, m)), \mathcal{R}_M)$                          // FIFO replay memory
        $\mathcal{B} \leftarrow$ set of $k$ tuples uniformly sampled from $\mathcal{R}_M$
        $\delta \leftarrow [0, \ldots, 0]$
        forall tuples $(s, m, s', U(s, m)) \in \mathcal{B}$ do
3           $y \leftarrow U(s, m) + \max_{m'}\{\hat{Q}_{\theta^-}(s', m')\}$
4           $\delta \leftarrow \delta + 1/k[y - Q_\theta(s, m)]\nabla_\theta Q_\theta(s, m)$
5       $\theta \leftarrow \theta + \alpha\delta$                              // update with Backpropagation
6       every $v$ steps reset $[\hat{Q}_{\theta^-}] \leftarrow [Q_\theta]$                  // update target net
```

Algorithm 16.1. CORL continual Online RL algorithm for RAE.

CORL (Continual Online RL) is a modified version of Deep Q-learning where actions are replaced with methods, and rewards are replaced with the utility function $U(s, m)$ as computed with Equation 15.1 at the end of a method run.[3]

Whenever RAE calls Guide for a new task or subtask, CORL records the returned method m (in Line 1). Recall that the choice of m by Guide results from calls to the planner UPOM. When RAE finishes executing m, the resulting state s' and observed reward $U(s, m)$ for this run are also recorded (in Line 2). The tuple $(s, m, s', U(s, m))$ is pushed in the replay memory $\mathcal{R}_M$, which is managed as a FIFO list recording the last N trials. A mini-batch $\mathcal{B}$ is sampled from $\mathcal{R}_M$; its records are used to update the parameter of $[Q_\theta]$ using a target network. The latter follows $[Q_\theta]$ with some delays.

Note that this is another instance of the "planning to learn" paradigm (Figure 1.2). The planner feeds CORL with initial training data. When online, RAE's calls to UPOM through Guide use the continually updated network $[Q_\theta]$ but do not immediately affect its updates. These are filtered out through what has been actually achieved. Only online runs by the actor in the real world are used for these updates.

[3]Equation 15.1 computes $U_{\text{stack}}(m, s)$ with reference to a current stack. We neglect the possible effects of other pending activities, as we did for Q_0.

An actor's runs may not be as frequent as wished for learning quickly. It is possible to augment actual experiences with simulations. However, these simulations should be based on problems (τ, s) that have actually been encountered, not on randomly generated ones. Moreover, the learning targets should not be the method-value function computed by UPOM, but the rewards $U(s, m)$ simulated with RAE with different methods. This simulation-augmented training makes it possible to broaden the learning on focused and realistic problems with inexpensive and safer simulated exploration while acting remains on cautious exploitation.

16.2 Learning Hierarchical Refinement Methods

A *hierarchical refinement domain* $\Sigma = (S, \mathcal{T}, \mathcal{M}, \mathcal{A})$ for RAE/UPOM can be seen globally as an MDP (see Section 14.3.5). Let us elaborate on this remark.

Learning policies for tasks. Assume that the set $\mathcal{T}$ of tasks and events in Σ is given as *annotated* tasks (see Section 7.2). Each $\tau \in \mathcal{T}$ can be formulated as an SSP problem from the set of states pre(τ) to the goal states eff(τ). Instead of seeking methods for a task τ, we can solve this SSP problem with the planning algorithms of Chapter 9 to obtain a good policy with respect to a cost or reward criteria. In a simulation-based approach with a Sample function, we do not need to estimate the probability distributions; we use Monte Carlo sampling techniques. An alternative to planning is to use reinforcement learning (see Chapter 10). We'll again end up with a policy, or equivalently, with a Q function. RL can be combined in a continual learning framework with acting.

These quite feasible approaches have a drawback: they give policies instead of hierarchical refinement methods. The latter are procedures with local variables, assignments and control structures, in addition to refinement and action steps. They are computationally more general than policies.

Learning control policies for methods. An approach that combines the advantages of refinement methods with learning is to use the idea of *partial programming* with hierarchical RL. A partial program specifies a program using the usual programming constructs, in addition to "choose" steps that offer open choices of a local variable within some ranges. A choose step is to be addressed with a *control policy* that maps the current state of the world and state of the program to an adequate choice for this variable. This control policy is to be learned with reinforcement learning.

The learner is given a refinement domain $\Sigma = (S, \mathcal{T}, \mathcal{M}, A)$ where:

- $\mathcal{T}$ is a set of annotated tasks;
- A is a set of primitive actions defined through a Sample function;
- $\mathcal{M}$ is a set of *partially specified* hierarchical refinement methods.

We are already familiar with the first two items; let us develop the latter. In addition to the usual primitive actions, task refinements, assignment and control steps, the body of a method in $\mathcal{M}$ has nondeterministic choose steps:

choose $x \in Range_x$, where x is a local variable and $range_x$ a finite set.

The learner has to synthesize a near-optimal deterministic control policy that gives the best value $x \in Range_x$ for the current world and control states.

Example 16.1. Consider the door-opening tasks of Example 14.4. The method m1-unlatch requires two constant values, a turning angle alpha1 and a pulling vector val1. We'd like to learn adequate values for this door. Furthermore, this method is rather coarse: either it succeeds on the first trial or it fails permanently. It would be good to learn if more than one trial is needed for this door.

```
mc-unlatch($r, d, l, o$)
   task: unlatch($r, d$)
    pre: location($r, l$)$\wedge$ toward-side($l, d$)$\wedge$ side($d$, left)$\wedge$ type($d$, rotate)
            $\wedge$ handle($d, o$)
  body: choose $k$ in $\{1, 2, 3\}$
        grasp($r, o$)
        while (door-status($d$) $\neq$ cracked or $k > 0$) do
            choose alpha1 in $\{.8, 1, 1.2\}$
            turn($r, o$, alpha1)
            choose val1 in $\{.1, .2, .3\}$
            pull($r$, val1)
            $k \leftarrow k - 1$
        if door-status($d$)=cracked then ungrasp($r, o$)
        else fail
```

The method mc-unlatch has three choose steps, for the number k of trials, the turning angle (in radians), and the pulling distance (in meters). We may also consider revising the method for the opendoor task in order to learn the features of a door d (i.e., its type, side and direction) instead of requiring it to be coded in the domain (see Exercise 16.1). □

With choose steps, $\mathcal{M}$ defines a hierarchy of partial programs for achieving the tasks in $\mathcal{T}$. A control policy giving good choices for those choose steps would allow RAE, to use the methods in $\mathcal{M}$ without the need of guidance from UPOM. Recall that UPOM was needed to choose among applicable alternative ground methods. Here, we can instead associate to each task τ a "root" method with a choose step over all the methods for τ, as illustrated next.

Example 16.2. Consider the DWR domain of Example 14.1 and 14.2. Assume that the navigate task can be achieved with three different strategies, depending on the area the robot is navigating in, e.g., with or without GPS localization, in a paved or dirt road, etc. The robot may have to switch from one strategy to another if its navigation takes it to a different area. We may discriminate among the navigation methods through their preconditions if the various areas in the environment are already well characterized and clearly delimited with respect to this robot capability. But adapting the environment to the robot is not a good idea. Alternatively, we may let the robot learn the best strategy to apply to its current area with the following "root" method:

```
mr-navigate($r, l$)
   task: navigate($r, l$)
    pre: location($r$) $\neq l$
   body: choose $nav$ in {strategy1, strategy2, strategy3}
         stepnav($r, l, nav$)
         navigate($r, l$)
```

Here *nav* is a variable characterizing the chosen navigation strategy, stepnav is a partial navigation task that uses the navigation strategy to go towards l. The methods for stepnav would stop after a few steps, depending on the strategy and the current state, to allow the robot to change its navigation strategy. □

Let us now discuss how to use hierarchical RL to learn a control policy for the partially specified methods of a domain $\Sigma = (S, \mathcal{T}, \mathcal{M}, A)$. We keep the presentation as informal as possible, but we need some additional notations:

- $\omega = (S, \mathcal{A})$ is the global MDP associated with Σ.
- π^*_ω is an optimal policy for MDP ω with the distributions implicit in Sample and a reward function (as in UPOM). The policy π^*_ω is independent of the methods in $\mathcal{M}$.
- $C_\mathcal{M}$ is the union of the ranges of all choose steps in $\mathcal{M}$. It is the set of control actions, each being a deterministic control for a chosen $x \in Range_x$. A control action leads to different states depending on the outcomes of the follow-up actions in A a refinement leads to.
- $\mathcal{E}_\mathcal{M}$ is the Cartesian product of the set of the world states S and the set of control states of $\mathcal{M}$. The latter set corresponds to the data structure stack of RAE/UPOM encoding which step in which method the control is. A choice in a choose step is a function on the current joint state in $\mathcal{E}_\mathcal{M}$.

Note that one may act in ω by following the methods in $\mathcal{M}$; let $\pi_{\omega/\mathcal{M}}$ be a policy for ω that follows $\mathcal{M}$. It can be demonstrated that:

- $\kappa = (\mathcal{E}_\mathcal{M}, C_\mathcal{M})$ is an MDP that corresponds to acting in ω by following the methods of $\mathcal{M}$.[4] κ is the control MDP for $\mathcal{M}$, with the distributions and reward functions of Σ, and zero reward for the control actions of $C_\mathcal{M}$.
- An optimal control policy π^*_κ for the MDP κ is an optimal policy $\pi_{\omega/\mathcal{M}}$ for ω that follows $\mathcal{M}$.

Consequently, if we are given partially specified hierarchical refinement methods, the domain learning problem is no longer about which action in $\mathcal{A}$ to perform next for the task at hand, i.e., for the MDP ω, but about the control policy for the choose steps in $\mathcal{M}$, i.e., for the MDP κ. The state space $\mathcal{E}_\mathcal{M}$ can be very large, but with

[4]More precisely κ is a Semi-Markov Decision Process. SMDP take into account that several states may occur between two consecutive actions. The difference is irrelevant when the discount factor is equal to one, for goal or task oriented MDPs.

reinforcement learning and the Sample function, we do not need to explicit κ. Here RL is hierarchical thanks to the guidance of the hierarchy of methods.

A simple approach for learning a control policy adapt the Q-learning algorithm for an adapted value function $Q : \mathcal{E}_{\mathcal{M}} \times C_{\mathcal{M}} \to \mathbb{R}$. Now, the update rule takes place between two consecutive choose steps when running $\mathcal{M}$ for a task τ. It is adapted from Equation 10.4 as follows:

$$Q(s,\sigma,x) \leftarrow \alpha[(\sum_{\sigma}^{\sigma'} r(a_i)) + \max_{x'}\{Q(s',\sigma',x')\}] + (1-\alpha)Q(s,\sigma,x) \qquad (16.1)$$

where x and x' are two control actions in two consecutive choose steps, σ and σ' are the respective states of the stack in these steps, s and s' the corresponding world states, the reward term is the sum of the rewards of all actions in $\mathcal{A}$ between the two steps. Recall that for the MDP κ, a control action x has zero reward, and the follow-up sequence of actions in $\mathcal{A}$ depends on $\mathcal{M}$ and the nondeterministic outcome of these actions.

Additional minor adaptations are needed in Q-learning:

- Line 1: select a control action x in a choose step;
- Line 2: follow $\mathcal{M}$ for a sequence of steps according to the choice of x until the next choose step;
- Line 3: accumulates the rewards between the two steps and observes the world and stack states.

Learning a control policy for $\mathcal{M}$ proceeds as follows:

- extract the task hierarchy from $\mathcal{M}$; order the set of tasks in $\mathcal{T}$ bottom-up, from the lowest level tasks in this hierarchy to the root tasks;
- for each $\tau \in \mathcal{T}$ in this bottom-up order, run the adapted Q-learning from random states in S_0 =pre(τ) to eff(τ) or to a failure termination.

The learned Q function defines the following control policy for the partially specified methods: $\pi_\kappa(s,\sigma) = \text{argmax}_x\{Q(s,\sigma,x)\}$.

To sum up, the learner in this approach gets more information than for learning policies for the global MDP ω for the tasks in $\mathcal{T}$ (see Section 14.3.5). In $\mathcal{M}$, it gets a task hierarchy as well as a guiding structure about which subtasks and actions can be relevant to which task and how. This can be seen as a specification burden. But the partially specified refinement methods are often naturally entailed from the definition of the tasks; they are easy to obtain and do not need a fine tuning thanks to the choose steps that are left open. Moreover, the approach has two advantages:

- it provides rich operational models with actions, refinements, local variables, assignments and control structures;
- it speeds up learning significantly giving good policies and enabling the transfer of learned tasks to others.

The basic Q-learning algorithm can be improved in a few ways, in particular to take into account the transitions in ω between two choose steps, which are not meaningful for κ. The resulting algorithm (called HAMQ-INT in the literature) brings additional learning speed-up. Other variants of the approach have been successfully developed. To our knowledge, none uses a parametric form of Q-learning nor neural nets. This is feasible and possibly important: the control action space $C_{\mathcal{M}}$ is generally small, but not the state space $\mathcal{E}_{\mathcal{M}}$.

16.3 Discussion and Bibliographic Notes

The basic techniques for this chapter are those of reinforcement learning and neural net estimators. Their references are discussed in Section 10.9.

The presented method for learning to guide RAE and UPOM is detailed [872]. This work compares the performances of different approaches, including for the choice of method parameters, on a few synthetic domains; it demonstrates significant efficiency and quality gains in the guidance of task refinements.

The technique for learning control policy for partially specified hierarchical refinement methods corresponds to the *Hierarchical Abstract Machine* (HAM) approach [870, 39, 756]. HAM proposes a programming language, ALisp, for partial programming with open choice steps which is modeled formally with finite state automata and SMDPs. The *Adaptive Behavior Language* approach of [1028] proposes a similar idea for writing adaptive software in game programming. The formal results mentioned here are proved in [870]. The HAMQ-INT algorithm using Moore automata is developed in [75]. Practical results are presented in these references, favorable comparisons to other hierarchical refinement approaches, such as [297], are given. An alternative similar approach to the HAM is proposed in [42] with *policy sketches*. Sketches annotate tasks with sequences of subtasks; the learner synthesizes with an actor–critic algorithm policies for subtasks maximizing the joint reward.

16.4 Exercises

16.1. Revise Example 14.4 with methods than can learn with choose steps the direction, side and type properties of the doors of a domain.

Part VI

Temporal Models

What then is time? If no one asks me, I know what it is. If I wish to explain it to him who asks, I do not know.

Augustine of Hippo, *Confessions* (Book 11, Chapter XIV), circa 398

An action may requite different kinds of resources, to be borrowed (e.g., space, tools) or consumed (e.g., energy). Time is a resource needed by every action, different from other resources. It flows independently from the actions being performed. It can be shared *ad infinitum* by independent actors as long as their actions do not interfere with each other.

In previous chapters, we left time implicit in our models: an action produced an instantaneous transition from one state to the next. However, deliberative acting often requires explicit temporal models. These models must specify when preconditions are required and when effects take place. For example, a robot moving from a location to a destination does not require the latter to be accessible at the outset but just shortly before it arrives.

Actions may, and sometimes must, overlap even if their conditions and effects are not independent. A robot may move from l to d while another one is concurrently moving from d to l. Opening a door that has a knob and a spring latch that controls the knob requires two tightly synchronized steps *(i)* pushing and maintaining the latch, *while (ii)* turning the knob. Modeling concurrency requires an explicit representation of time. Additional motivations for explicit time are about goals constrained with deadlines, or events expected to occur at future time periods, for example, the arrival of scheduled ships at a harbor. Actions may have to be located in time with respect to expected events or deadlines. Time can be required *qualitatively*, to handle synchronization with actions and events, and *quantitatively*, to model the duration of actions with respect to various parameters.

In summary, the motivations for making time explicit are the following:

- modeling the duration of actions;

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- modeling the effects, conditions, and resources borrowed or consumed at various moments along an action duration, and its delayed effects;
- handling concurrent actions that have interacting and joint effects;
- handling goals with relative or absolute temporal constraints;
- planning and acting with respect to exogenous events that are expected to occur at a future time; and
- planning with actions that maintain a value while being executed, as opposed to just changing that value (e.g., tracking a moving target, or keeping a spring latch in some position).

This part of the book is about planning, acting, and learning approaches in which time is explicit. It describes several algorithms and methods for handling durative and concurrent activities with respect to a predicted dynamics. It is interesting to note that acting with temporal models raises dispatching and temporal controllability issues that heavily rely on planning concepts. Because of that, we depart here from the outline of other parts in this book with the planning chapter before the acting one.

Chapter 17 introduces a knowledge representation for modeling actions and tasks with temporal variables and temporal refinement methods, an extension of the methods seen earlier. Temporal planning problems and temporal plans are defined as *chronicles*, that is, collections of assertions and tasks with explicit temporal constraints. A planning algorithm with temporal refinement methods is developed in Section 17.2. The basic techniques for managing temporal and domain constraints are then presented in Section 17.3.

Chapter 18 considers the dynamic controllability of a temporal plan at acting time and presents a dispatching algorithm. Acting problems and methods with temporal domain models are then discussed for different types of operational models. Finally, Chapter 19 addresses learning heuristics for temporal planning and learning temporal action and task models.

17 Temporal Representation and Planning

This chapter is about planning approaches with explicit time in the descriptive and operational models of actions, as well as in the models of the expected evolution of the world not caused by the actor. It describes a planning algorithm that handles durative and concurrent activities with respect to a predicted dynamics.

Section 17.1 presents a knowledge representation for modeling actions and tasks with temporal variables using temporal refinement methods. Temporal plans and planning problems are defined as *chronicles*, that is, collections of assertions and tasks with explicit temporal constraints. A planning algorithm with temporal refinement methods is developed in Section 17.2. The basic techniques for managing temporal and domain constraints are then presented in Section 17.3.

17.1 Temporal Representation

An representation of time for planning and acting can be either:

- "*State-oriented*": one keeps the notion of global states of the world, as we have done so far, and includes time explicitly in the model of the transitions between states (e.g., as in timed automata). The dynamics of the world is modeled as a collection of global snapshots, each of which gives a complete description of the domain at some time point.
- "*Time-oriented*": one represents the dynamics of the world as a collection of partial functions of time, describing local evolutions of state variables. Instead of a state, the building block here is a *timeline* (horizontal slice in Figure 17.1) that focuses on one state variable and models its evolution in time. Time-oriented approaches use as actions either instants or intervals , with qualitative and/or quantitative relations.

We use here a time-oriented approach with quantitative relations on time points (see Section 18.7 for a discussion of state-oriented approaches). The proposed representation relies on *timelines* and temporal assertions to model actions, tasks, methods and *chronicles*.

17.1.1 Assertions and Timelines

A *quantitative discrete model of time* is described by a collection of temporal variables, $t, t', t_1, t_2, \ldots$; each variable designates a time point. For simplicity, we assume that temporal variables range over the set of integers.[1] An *interval* is a pair $[t, t']$ such

[1]This assumption avoids some minor issues regarding closed versus open intervals.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

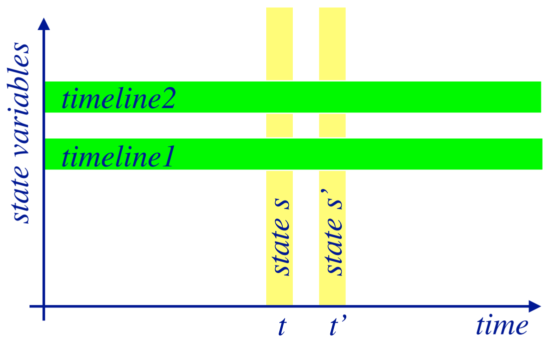


Figure 17.1. State-oriented versus time-oriented views.

that $t < t'$; its duration is $t' - t > 0$. We also use open intervals, for example, $[t, t')$, meaning in our case $[t, t' - 1]$.

Temporal variables are not instantiated at planning time into numeric values. They are constrained with respect to other temporal variables or constants. Temporal constraints are specified with the usual arithmetic operators ($<, \leq, =$, etc.) between temporal variables and integer constants, e.g., $t < t'$ says that t is before t'; $d \leq t' - t \leq d'$ constrains the duration of the interval $[t, t']$ between the two bounds d and d'. These constraints have to be maintained consistent. The value of a temporal variable is set at acting time when actions and observations are performed. To sum up, a temporal variable remains constrained but uninstantiated as long as it refers to the future. It is instantiated with a value corresponding to the *current time* when the fact this variable qualifies takes place, either controlled or observed by the actor. After that point, the variable refers to the past. Each state variable x is a function of time. The evolution of its value over time is represented with temporal assertions.

Definition 17.1. A *temporal assertion* on a state variable x is either a persistence or a change:

- A *persistence*, denoted $[t_1, t_2]\, x = v$, specifies that x has the value v over the interval $[t_1, t_2]$.
- A *change*, denoted $[t_1, t_2]\, x : (v_1, v_2)$, specifies that the value of x changes over the interval $[t_1, t_2]$ from v_1 at t_1 to v_2 at t_2, with $v_1 \neq v_2$. □

As a shorthand, $[t]\, x = v$ stands for $[t, t + 1]\, x = v$, and $[t]\, x : (v, v')$ stands for $[t, t + 1]\, x : (v, v')$. The former gives the value of x at a single time point and the latter expresses a transition from v to v' over two consecutive time-points. In general, and assertion $[t, t']\, x : (v, v')$ does not model how the change takes place within the interval $[t, t']$; it can be gradual over possibly intermediate values or instantaneous at any moment in $[t, t']$. However, if $t' = t + 1$, then the value of x changes discretely from v at time t to v' at time $t + 1$.

For example, the assertion $[t_1, t_2]$ loc(r1) : (loc2,loc3) says that r1's location changes from loc2 to loc3. The precise moments of this change and intermediate values of loc(r1) are not stated by this assertion. Their values will be established by the command that performs the change from loc2 to loc3.

Temporal assertions can be parameterized, for example, $[t_1, t_2]\, \mathsf{loc}(r) : (l, \mathsf{loc1})$ states that some robot r moves from a location l to loc1. The values of r and l will be fixed at some planning or acting stage; the values of t_1 and t_2 are instantiated only at acting time.

Definition 17.2. A *timeline* is a pair $(\mathcal{T}, C)$ where $\mathcal{T}$ is a conjunction of temporal assertions on a state variable, possibly parameterized with object variables, and C is a conjunction of constraints on the temporal variables and the object variables of the assertions in $\mathcal{T}$. □

A temporal planner maintains and updates timelines by adding new assertions or constraints, and testing their consistency properties, explained next. Definition 17.2 allows for parameterized state variables. For example $\mathcal{T}$ may contain assertions referring to loc(r1) and loc(r), which (depending on r) may refer either to one or two timelines. We assume $\mathcal{T}$ to refer to a single timeline as long as $r \neq \mathsf{r1}$ cannot be entailed from C. Hence a timeline may be separated into several timelines if instantiation constraints are added to C.

$\mathcal{T}$ and C are denoted as sets of assertions and constraints. Constraints on temporal variables are unary and binary inequalities and equalities. If $\mathcal{T}$ contains a persistence $[t_1, t_2]\, x = v$ or a change $[t_1, t_2]\, x : (v_1, v_2)$, then the constraint $t_1 \leq t_2$ is in C, implicit for ease of notation. Constraints on object variables are with respect to rigid relations, for example, connected(l,loc1), or binding constraints, as in the following example.

Example 17.3. The whereabouts of the robot r1, as depicted in Figure 17.2, can be expressed with the following timeline:

$$(\{[t_1, t_2]\, \mathsf{loc(r1)} : (\mathsf{loc1}, l),\ [t_2, t_3]\, \mathsf{loc(r1)} = l,\ [t_3, t_4]\, \mathsf{loc(r1)} : (l, \mathsf{loc2})\},$$
$$\{t_1 < t_2 < t_3 < t_4,\ l \neq \mathsf{loc1},\ l \neq \mathsf{loc2}\})$$

In this timeline, $\mathcal{T}$ has three assertions: one persistence and two changes; C has temporal and object constraints. The constraints are in this particular case entailed from the three intervals and two change assertions in $\mathcal{T}$. Instances of the timeline are substitutions of possible values in these assertions for the five variables $l, t_1, \ldots, t_4$.

Note that this timeline does not say what happens between t_1 and t_2; all we know is that r1 leaves loc1 at or after t_1, and it arrives at l at or before t_2. To say that these two changes happen exactly at t_1 and t_2, we can add the following assertions in the timeline: $[t_1, t_1 + 1]\, \mathsf{loc(r1)} : (\mathsf{loc1}, \mathsf{route})$ and $[t_2 - 1, t_2]\, \mathsf{loc(r1)} : (\mathsf{route}, l)$, where route is some intermediate location. These assertions say that $[t_1]\, \mathsf{loc(r1)} = \mathsf{loc1}$, $[t_1 + 1]\, \mathsf{loc(r1)} = \mathsf{route}$, $[t_2 - 1]\, \mathsf{loc(r1)} = \mathsf{route}$, and $[t_2]\, \mathsf{loc(r1)} = l$. □

Temporal assertions in a timeline $(\mathcal{T}, C)$ are expressed with temporal and object variables that can be instantiated within their respective domains with the usual unification mechanisms. Not every instance of a timeline makes sense as a possible consistent evolution. A pair of temporal assertions on a state variable x is *conflicting* if it can have two different values for x at the same time; otherwise, it is nonconflicting. Since change assertions do not give the precise point at which a change occurs, the two assertions $[t_1, t_2]\, x : (v_1, v_2)$ and $[t'_1, t'_2]\, x : (v'_1, v'_2)$ are conflicting if they overlap

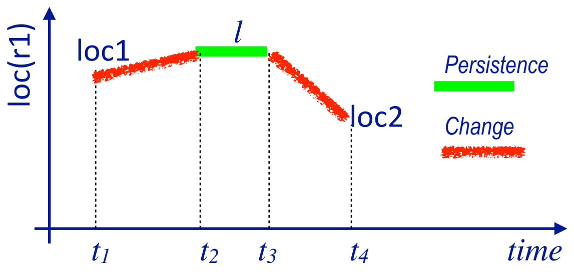


Figure 17.2. A timeline for the state variable loc(r1). The positions of the points on the two axes are qualitative; the rough lines do not necessarily represent linear changes.

in time, unless they are strictly identical, or the overlap is only at their endpoints, i.e., $v_2 = v'_1$ and $t_2 = t'_1$, or $v'_2 = v_1$ and $t'_2 = t_1$.

Definition 17.4. A ground instance of $(\mathcal{T}, C)$ is *consistent* if it satisfies all the constraints in C and does not have a pair of conflicting assertions. A timeline $(\mathcal{T}, C)$ is *consistent* if its set of consistent instances is not empty. □

A *separation constraint* for a pair of conflicting assertions is a conjunction of constraints on object and temporal variables that exclude inconsistent instances. The set of separation constraints for a conflicting pair of assertions contains all possible conjunctions that exclude inconsistent instances.

Example 17.5. The two persistence assertions $\{[t_1, t_2]\, \text{loc}(r) = \text{loc1}, [t_3, t_4]\, \text{loc(r1)} = l\}$ are conflicting, because they can have inconsistent instances. For example, if $r = \text{r1}, l \neq \text{loc1}$ and either $t_1 \leq t_3 \leq t_2$ or $t_1 \leq t_4 \leq t_2$, then the robot r1 would have to be at loc1 and at $l \neq \text{loc1}$ simultaneously.

The pair of assertions $\{[t_1, t_2]\, \text{loc(r1)} = \text{loc1}, [t_2, t_3]\text{loc(r1)} : (\text{loc1}, \text{loc2})\}$ is nonconflicting: they have no inconsistent instances.

The pair $\{[t_1, t_2]\, \text{loc(r1)} = \text{loc1}, [t_3, t_4]\, \text{loc(r1)} : (l, l')\}$ is conflicting. A separation constraint is $(t_2 = t_3, l = \text{loc1})$.

The set of separation constraints for that pair is:
$\{(t_2 < t_3), (t_4 < t_1), (t_2 = t_3, l = \text{loc1}), (t_4 = t_1, l' = \text{loc1})\}$. □

A set of assertions is conflicting if any pair of the set is. A separation constraint for a set of conflicting assertions is a consistent conjunction of constraints that makes every pair of the set nonconflicting. A set of assertions may have separation constraints for every pair while there is no consistent conjunction of separation constraints for the entire set.

Example 17.6. Consider the set of assertions $\{[t_1, t_2]\, \text{loc(r1)} : (\text{loc1}, \text{loc2}), [t_2, t_3]\, \text{loc(r1)} = l, [t_3, t_4]\, \text{loc(r1)} : (\text{loc3}, \text{loc4})\}$. The constraint $l = \text{loc2}$ is a separation for the first two assertions, while the constraint $l = \text{loc3}$ is required for the last two assertion. □

The consistency of a timeline $(\mathcal{T}, C)$ is a stronger notion than just satisfying the constraints in C. It also requires the assertions in $\mathcal{T}$ to have a nonconflicting instance that satisfies C. A timeline is inconsistent if in particular there are no separation constraints, or none that is consistent with C. A convenient case is when C includes the separation constraints needed by $\mathcal{T}$. For such a case, satisfying the constraints in C guarantees the consistency of the timeline. This is the notion of secure timelines.

Definition 17.7. A timeline $(\mathcal{T}, C)$ is *secure* if and only if it is consistent and every instance that meets the constraints in C is consistent. □

In a secure timeline $(\mathcal{T}, C)$, no instance that satisfies C specifies different values for the same state variable at the same time. In other words, every pair of assertions in $\mathcal{T}$ is either nonconflicting or has a separation constraint entailed from C. A consistent timeline may possibly be augmented with separation constraints to make it secure.

Example 17.8. The timeline $(\{[t_1, t_2]\, \text{loc(r1)} = \text{loc1},\ [t_3, t_4]\, \text{loc(r1)} : (\text{loc1}, \text{loc2})\}, \{t_2 < t_3\})$ is secure; its assertions are nonconflicting. The timeline $(\{[t_1, t_2]\, \text{loc}(r) = \text{loc1},\ [t_3, t_4]\, \text{loc(r1)} = l\}, \{t_1 < t_2,\ t_3 < t_4\})$ is consistent but not secure; when augmented with either $(r \neq \text{r1})$ or $(t_2 < t_3)$ it becomes secure. □

Another important notion is that of the *causal support* of an assertion in a timeline. Timelines are used to reason about the dynamic evolution of a state variable. An actor's reasoning about a timeline requires every element in this evolution to be either given by its observation or prior knowledge (e.g., for the initial state), or explained by some reason due the actor's own actions or to the dynamics of the environment. For example, looking at the timeline in Figure 17.2, the locations of the robot in l, then in loc2, are explained by the two change assertions in that timeline. However, nothing explains how the robot got to loc1; we have to state an assertion saying that it was there initially or brought there by a move action.

Definition 17.9. An assertion $[t, t']\, x = v$ or $[t, t']\, x : (v, v')$ in a timeline is *causally supported* if the timeline contains another assertion $[t'', t]\, x = v$ or $[t'', t]\, x : (v'', v)$ that asserts the value v at time t. □

Note that by definition of the intervals $[t'', t]$ and $[t, t']$ we have $t'' < t < t'$. Hence this definition excludes circular support, that is, assertion α cannot support assertion β while β supports α, regardless of whether this support is direct or by transitivity via some other assertions.

Example 17.10. In Example 17.3 assertion $[t_2, t_3]\, \text{loc(r1)} = l$ is supported by $[t_1, t_2]\, \text{loc(r1)} : (\text{loc1}, l)$. Similarly, assertion $[t_3, t_4]\, \text{loc(r1)} : (l, \text{loc2})$ is supported by $[t_2, t_3]\, \text{loc(r1)} = l$. However, the first assertion in that timeline is unsupported: nothing asserts $[t_1]\, \text{loc(r1)} = \text{loc1}$. □

It may be possible to support an assertion in a timeline by adding constraints on object and temporal variables. For example, $[t_1, t_2]\, \text{loc(r1)} : (\text{loc1,loc2})$ can be supported by $[t, t']\, \text{loc}(r) = l$ if the following constraints are added to the timeline: $(t' = t_1, \text{r=r1}, l = \text{loc1})$. Another way of supporting an assertion is by adding a persistence condition. For example, in the timeline $(\{[t_1, t_2]\text{loc(r1)} : (\text{loc1,loc2}), [t_3, t_4]\, \text{loc(r1)}:(\text{loc2,loc3})\}, \{t_1 < t_2 < t_3 < t_4\})$, the second assertion can be supported by adding the following persistence: $[t_2, t_3]$ loc(r1) = loc2. Adding a change assertion can also be used to support assertions. As we'll see in Section 17.2.3, adding a new action to a plan results in new assertions that can provide the required support.

Let us extend to sets of timelines the previous definitions. If $\mathcal{T}$ is a set of temporal assertions on several state variables and C is a set of constraints, then the pair $(\mathcal{T}, C)$ corresponds to a set of timelines $\{(\mathcal{T}_1, C_1), \ldots, (\mathcal{T}_k, C_k)\}$. $(\mathcal{T}, C)$ is consistent or secure if each of its timelines is.

While reasoning about actions and their effects, an actor will perform the following operations on a set of timelines $(\mathcal{T}, C)$:

- add constraints to C, to secure a timeline or support its assertions; for example, for the first timeline in Example 17.8, the constraint $t_2 = t_3$ makes the assertion $[t_3, t_4]$ loc(r1) : (loc1, loc2) supported.
- add assertions to $\mathcal{T}$, for example, for the timeline in Figure 17.2 to take into account additional motions of the robot.
- instantiate some of the variables, which may possibly split a timeline of the set with respect to different state variables, for example, assertions related to loc(r) and loc(r') refer to the same state variable, but that timeline will be split if r is instantiated as r1 and r' as r2.

17.1.2 Actions

An action is modeled as a collection of timelines. More precisely, an action schema is a triple $(head, \mathcal{T}, C)$, where *head* is the name and arguments of the action, and $(\mathcal{T}, C)$ is a set of timelines. This representation is an extension of the action schemas of Chapter 2 with explicit time expressing conditions and effects at different moments during the time span of an action.

Example 17.11. Suppose several robots are moving in a connected network of roads servicing loading docks. Fixed in each dock are one crane and several piles where containers are stacked. A dock can contain at most one robot at a time. Robots and cranes can carry at most one container at a time. Waypoints in roads guide the robot navigation; two waypoints are connected if there is a path through the road network between them.

The objects in this domains are: $r \in Robots$, $k \in Cranes$, $c \in Containers$, $p \in Piles$, $d \in Docks$, $w \in Waypoints$.

The invariant structure of the domain is given by three rigid relations:

$$\begin{aligned}
&\text{attached} \subseteq (\mathit{Cranes} \cup \mathit{Piles}) \times \mathit{Docks}\\
&\text{adjacent} \subseteq \mathit{Docks} \times \mathit{Waypoints}\\
&\text{connected} \subseteq \text{Waypoints} \times \text{Waypoints}
\end{aligned}$$

The domain is described with the following state variables:

$$\begin{aligned}
\text{loc}(r) &\in \mathit{Docks} \cup \mathit{Waypoints} &&\text{for } r \in \mathit{Robots}\\
\text{freight}(r) &\in \mathit{Containers} \cup \{\text{empty}\} &&\text{for } r \in \mathit{Robots}
\end{aligned}$$

$$
\begin{aligned}
\mathsf{grip}(k) &\in \mathit{Containers} \cup \{\mathsf{empty}\} && \text{for } k \in \mathit{Cranes}\\
\mathsf{pos}(c) &\in \mathit{Robots} \cup \mathit{Cranes} \cup \mathit{Piles} && \text{for } c \in \mathit{Containers}\\
\mathsf{stacked\text{-}on}(c) &\in \mathit{Containers} \cup \{\mathsf{empty}\} && \text{for } c \in \mathit{Containers}\\
\mathsf{top}(p) &\in \mathit{Containers} \cup \{\mathsf{empty}\} && \text{for } p \in \mathit{Piles}\\
\mathsf{occupant}(d) &\in \mathit{Robots} \cup \{\mathsf{empty}\} && \text{for } d \in \mathit{Docks}.
\end{aligned}
$$

The constant empty means that a robot, a crane, a pile, or a dock is empty, or that a container is not stacked on any other container.

The task in this example is to bring containers from their current position to a destination pile. It is specified with actions, tasks, and methods (to which we'll come back in the next section). The actions are the following:

$$
\begin{aligned}
\mathsf{leave}(r,d,w) &: \text{robot } r \text{ leaves dock } d \text{ to an adjacent waypoint } w,\\
\mathsf{enter}(r,d,w) &: r \text{ enters } d \text{ from an adjacent waypoint } w,\\
\mathsf{navigate}(r,w,w') &: r \text{ navigates from waypoint } w \text{ to } w',\\
\mathsf{stack}(k,c,p) &: \text{crane } k \text{ holding container } c \text{ stacks it on top of pile } p,\\
\mathsf{unstack}(k,c,p) &: \text{crane } k \text{ unstacks a container } c \text{ from the top of pile } p,\\
\mathsf{put}(k,c,r) &: \text{crane } k \text{ holding a container } c \text{ and puts it onto } r,\\
\mathsf{take}(k,c,r) &: \text{crane } k \text{ takes container } c \text{ from robot } r.
\end{aligned}
$$

A descriptive model of leave is specified by the following schema:

$\mathsf{leave}(r,d,w)$
- assertions: $[t_s,t_e]\ \mathsf{loc}(r):(d,w)$
  $[t_s,t_e]\ \mathsf{occupant}(d):(r,\mathsf{empty})$
- constraints: $t_e \le t_s + \delta_1$
  $\mathsf{adjacent}(d,w)$

This expression says that the leave action changes the location of r from dock d to the adjacent waypoint w, with a delay smaller than δ_1 after the action starts at t_s; the dock d is empty when the action ends at t_e.

Similarly, enter is defined by the following action schema:

$\mathsf{enter}(r,d,w)$
- assertions: $[t_s,t_e]\ \mathsf{loc}(r):(w,d)$
  $[t_s,t_e]\ \mathsf{occupant}(d):(\mathsf{empty},r)$
- constraints: $t_e \le t_s + \delta_2$
  $\mathsf{adjacent}(d,w)$

The take action is specified as follows:

$\mathsf{take}(k,c,r)$
- assertions: $[t_s,t_e]\ \mathsf{pos}(c):(r,k)$
  $[t_s,t_e]\ \mathsf{grip}(k):(\mathsf{empty},c)$
  $[t_s,t_e]\ \mathsf{freight}(r):(c,\mathsf{empty})$
  $[t_s,t_e]\ \mathsf{loc}(r)=d$
- constraints: $\mathsf{attached}(k,d)$, $\mathsf{attached}(p,d)$

The assertions in this action say that a container c loaded on r at t_s is taken by crane k at t_e; r remains in the same dock as k.

Similar specifications are required for the actions put(k, c, r), to put a container on r, stack(k, c, p), to put the container c held by k on top of pile p, unstack(k, c, p), to take with k the top container c of pile p, and navigate(r, w, w') to navigate between connected waypoints (see Exercise 17.2).

Note that actions leave, enter, take, and so on, are said to be action at the planning level, but they will be refined at the acting level. We'll see in Example 18.8 how to further refine them into executable commands. □

As illustrated in Example 17.11, actions are specified as assertions and constraints on temporal variables and object variables. By convention, t_s and t_e denote the starting point and ending point of each action. The temporal variables of an action schema are not in its list of parameters because we are going to handle them differently from the object variables. The planner will instantiate object variables, but it will only constrain the temporal variables with respect to other time points. Their instantiation into constants is performed at acting time, from the triggering of actions and observation of events (as controllable and uncontrollable time points, see Chapter 18).

This representation does not use two separate fields for preconditions and effects. A change in a action, such as $[t_s, t]$ grip(k) : (empty, c), expresses both the precondition that crane k should be empty at time t_s and the effect that k holds container c at time t. The temporal assertions in a action refer to several instants, not necessarily ordered, within the timespan of an action.

Temporal and object variables in a action are free variables. To make sure that different instances of a action, say take, refer to different variables t_s, t_e, k, r, c, we rely on the usual variable renaming, to be detailed later.

17.1.3 Methods and Tasks

A task is a label naming an activity to be performed and temporal qualification, written as : $[t, t']$*task*. This expression means that *task* starts at or after t, and finishes at or before t'. It does not require *task* to persist throughout the entire interval (contrarily to a persistence condition on a state variable).

A task is refined into subtasks and actions using *temporal refinement methods*. A method is a tuple (*head, task, refinement,* $\mathcal{T}, \mathcal{C}$), where *head* is the name and arguments of the methods, *task* gives the task to which the method applies, *refinement* is the set of temporally qualified subtasks and actions in which it refines *task*, $\mathcal{T}$ are assertions and $\mathcal{C}$ constraints on temporal and object variables. A temporal refinement method does not need a separate precondition field, as in the methods of previous chapter. This is because temporal assertions may express conditions as well as effects in a flexible way and at different moments, as illustrated next.

Example 17.12. The task of bringing containers to destination piles in Example 17.11 can be modeled with the following tasks: bring, move, uncover, load, and unload. A possible method for bring is:

```
m-bring$(r, c, p, p', d, d', k, k')$
          task: bring$(r, c, p)$          # $r$ brings container $c$ to pile $p$
    refinement: $[t_s, t_1]$ move$(r, d')$
                $[t_s, t_2]$ uncover$(c, p')$
                $[t_3, t_4]$ load$(k', r, c, p')$
                $[t_5, t_6]$ move$(r, d)$
                $[t_7, t_e]$ unload$(k, r, c, p)$
    assertions: $[t_s, t_3]$ pile$(c) = p'$
                $[t_s, t_3]$ freight$(r) =$ empty
   constraints: attached$(p', d')$, attached$(p, d)$, $d \neq d'$
                attached$(k', d')$, attached$(k, d)$
                $t_1 \leq t_3, t_2 \leq t_3, t_4 \leq t_5, t_6 \leq t_7$
```

This method refines bring into five subtasks to move the robot to d' then to d, to uncover container c to have it at the top of pile p', to load the robot in d' and unload it in d in the destination pile p. As depicted in Figure 17.3, the first move and uncover are concurrent (t_2 and t_3 are unordered). When both tasks finish, the remaining tasks are sequential. Container c remains in its original pile, and robot r remains empty until the load task starts.

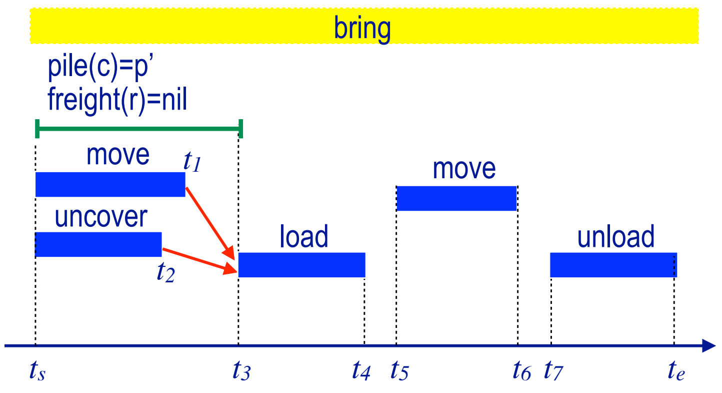


Figure 17.3. Assertions, actions and subtasks of a refinement method for the bring task. The diagonal arrows represent precedence constraints.

```
m-move1$(r, d, d', w, w')$
          task: move$(r, d)$          #moves a robot $r$ to a dock $d$
    refinement: $[t_s, t_1]$ leave$(r, d', w')$
                $[t_2, t_3]$ navigate$(r, w', w)$
                $[t_4, t_e]$ enter$(r, d, w)$
    assertions: $[t_s, t_s + 1]$ loc$(r) = d'$
   constraints: adjacent$(d, w)$, adjacent$(d', w')$, $d \neq d'$
                connected$(w, w')$
                $t_1 \leq t_2, t_3 \leq t_4$
```

This method refines the move to a destination dock d into three successive steps: leave the starting dock d' to an adjacent waypoint w', navigate to a connected waypoint w adjacent to the destination and enter the destination d, which is required to be empty only when the robot gets there. The move task requires additional methods to address cases in which the robot starts from a road or when it is already there (see Exercise 17.3).

```
m-uncover$(c, p, k, d, c', p')$
          task: uncover$(c, p)$        #un-pile $p$ until its top is $c$
    refinement: $[t_s, t_1]$ unstack$(k, c', p)$
                $[t_2, t_3]$ stack$(k, c', p')$
                $[t_4, t_e]$ uncover$(c, p)$
    assertions: $[t_s, t_s + 1]$ pile$(c) = p$
                $[t_s, t_s + 1]$ top$(p) = c'$
                $[t_s, t_s + 1]$ grip$(k) =$ empty
   constraints: attached$(k, d)$, attached$(p, d)$,
                attached$(p', d)$, $p \neq p'$, $c' \neq c$
                $t_1 \leq t_2, t_3 \leq t_4$
```

This method refines uncover into unstacking the container at the top of pile p, moving it to a nearby pile p' and then invoking uncover again if the top of p is not c. Another method should handle the case where c is at the top of p. Task load can be refined into action unstack and put; task unload is similarly refined into take and stack (see Exercise 17.3). □

Assertions in methods specify conditions as well as effects at any moment during the duration of the task. Note that the specific assertions conditioning the subtasks and actions of a task τ should be expressed in their respective definitions, not in the methods handling task τ. Redundancy between conditions in methods, and conditions in subtasks and actions is not desirable. For example, the action enter has the assertion $[t_s, t_e]$ occupant(d) : (empty, r); the same assertion (with different variables that will be unified with t_s, t_e, d and r) may appear in the method m-move1, but it is not needed. Redundancy, as well as incomplete specifications, are sources of errors.

Planning and acting procedures will view tasks as labelled networks with associated contraints. For example, a task bring in Example 17.12 can be the root of a task network whose first successor with method m-bring is a task move, which in turn leads with m-move1 to the action leave. A leaf in a task network is a action. An inner node is a task, which, at some point in the planning and/or acting process, is either:

- *refined* : it is associated with a method; it has successors labelled by subtasks and actions as specified in the method with the associated constraints; or
- *unrefined*: its refinement with an applicable method is pending.

The refinement mechanism takes place with a data structure called a chronicle.

17.1.4 Chronicles

A *chronicle* expresses a planning problem or a partial plan. It is a collection of temporally qualified tasks, actions, and assertions with associated constraints. It specifies:

(i) the tasks to be performed;
(ii) the *a priori* given and known facts about the current state and predicted future that will take place independently of the planned activities; and
(iii) the assertions to be achieved, which are constraints on future states of the world that planning will have to satisfy.

The elements in (ii) are also expressed as temporal assertions, we refer to them as *the* a priori *known supported* assertions to distinguish them from assertions in (iii), which require support from the planned activities. More formally:

Definition 17.13. A chronicle is a tuple $\phi = (\mathcal{A}, \mathcal{K}, \mathcal{T}, \mathcal{C})$ where $\mathcal{A}$ is a set of temporally qualified actions and tasks, $\mathcal{K}$ is a set of *a priori* known supported assertions, $\mathcal{T}$ is a set of assertions, and $\mathcal{C}$ is a conjunction of constraints on the temporal and object variables in $\mathcal{A}, \mathcal{K}$, and $\mathcal{T}$. □

Example 17.14. Assume in Example 17.12 that a pile p can be on a ship, and that a crane k on a dock d can unstack containers from that pile p only when the corresponding ship is docked at d (see Exercise 17.4).

Consider the case in which this domain has two robots r1 and r2, initially in dock1 and dock2, respectively. A ship ship1 is expected to be docked at dock3 at a future interval of time; it has a pile, pile-ship1, the top element of which is a container c1. The problem is to bring container c1 to dock4 using any robot and to have the two robots back at their initial locations at the end. This problem is expressed with the following chronicle:

$$
\begin{array}{rl}
\phi_0: & \\
\text{tasks:} & [t, t'] \,\mathsf{bring}(r, \mathsf{c1}, p) \\
\text{supported:} & [t_s] \,\mathsf{loc(r1)=dock1} \\
 & [t_s] \,\mathsf{loc(r2)=dock2} \\
 & [t_s] \,\mathsf{top(pile\text{-}ship1)=c1} \\
 & [t_s] \,\mathsf{pos(c1)=pallet} \\
 & [t_s + 10, t_s + \delta] \,\mathsf{docked(ship1)=dock3} \\
\text{assertions:} & [t_e] \,\mathsf{loc(r1) = dock1} \\
 & [t_e] \,\mathsf{loc(r2) = dock2} \\
\text{constraints:} & t_s < t < t' < t_e, 20 \leq \delta \leq 30, t_s = 0 \\
 & \mathsf{attached}(p, \mathsf{dock4})
\end{array}
$$

t_s and t_e denote the starting and end points of a chronicle. The planning problem is specified by taking t_s the origin of the clock. Here t_s is grounded with respect to the current time for the predicted future given in the last supported assertion. The goal of bringing c1 to dock4 is expressed through the task bring(r, c1,p) and the constraint attached(p, dock4). □

Chronicles also express partial plans that will be progressively transformed by a planner into complete solution plans.

Example 17.15. Consider the two robots r1 and r2 of Example 17.12 performing concurrent actions where each robot moves from its dock to the other robot's dock as depicted in Figure 17.4. The following chronicle (where $\mathcal{K}$ and $\mathcal{T}$ are not detailed) expresses this set of coordinated actions:

ϕ :

tasks: $[t_0, t_1]$ leave(r1,dock1,w1)
$[t_1, t_2]$ navigate(r1,w1,w2)
$[t_3, t_4]$ enter(r1,dock2,w2)
$[t'_0, t'_1]$ leave(r2,dock2,w2)
$[t'_1, t'_2]$ navigate(r2,w2,w1)
$[t'_3, t'_4]$ enter(r2,dock1,w1)
supported: $\mathcal{K}$
assertions: $\mathcal{T}$
constraints: $t'_1 < t_3, t_1 < t'_3, t_s < t_0, t_s < t'_0, t_4 < t_e, t'_4 < t_e$
adjacent(dock1,w1), adjacent(dock2,w2)
connected(w1,w2)

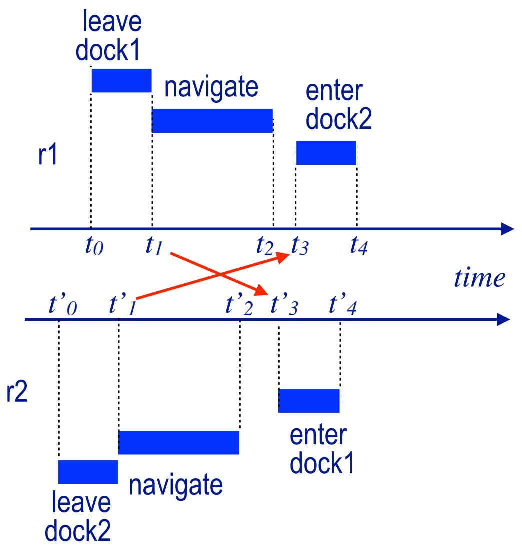


Figure 17.4. Temporally qualified actions of two robots, r1 and r2. The diagonal arrows represent the precedence constraints $t'_1 < t_3$ and $t_1 < t'_3$.

This chronicle says that r1 leaves dock1 before r2 enters dock1 ($t_1 < t'_3$); similarly, r2 leaves dock2 before r1 gets there ($t'_1 < t_3$). Each action navigate starts when the corresponding leave finishes (t_1 and t'_1). However, an enter may have to wait until the navigate finishes (t_2 to t_3) and the way is free. □

The set $\mathcal{T}$ of assertions in a chronicle $\phi = (\mathcal{A}, \mathcal{K}, \mathcal{T}, C)$ contains all the assertions of the actions already in $\mathcal{A}$, for example, leave and enter in Example 17.15. When a task $\tau \in \mathcal{A}$ is refined with a method m, τ is replaced in $\mathcal{A}$ by the subtasks and actions specified in m, and $\mathcal{T}$ and C are augmented with the assertions and constraints of m and those of its actions.

When a task is refined, the free variables in methods and actions are renamed and possibly instantiated. For example, enter is specified in Example 17.11 with the free variables r, d, w, t_s, t_e. In the first instance of enter in the chronicle of Example 17.15, these variables are respectively bound to r1, dock2, w2, t_3, and t_4. In the second instance of enter, they are bounded to r2, dock1, w1, t'_3, t'_4. The general mechanism for every instance of a action or a method is to rename the free variables in its schema to new names, then to constrain and/or instantiate these renamed variables when needed.

When refining a task and augmenting the assertions and contraints of a chronicle, as specified by a method, we need to make sure that $(\mathcal{T}, C)$ remains secure. Separation constraints will be added to C to handle conflicting assertions. A planner has to maintain the consistency of the resulting constraints and to support all the assertions of a chronicle.

17.2 A Hybrid Temporal Planner

This section presents TemPlan, a temporal planner for addressing problems defined with tasks to be performed as well as goals to be reached, in addition to what the tasks achieves. TemPlan performs *refinement planning* (as in HTN) and *generative planning* (as in plan-space planning). For this reason, we qualify it as a *hybrid* planner

A temporal planning domain Σ is defined by giving the sets of objects, rigid relations and state variables of the domain, and by specifying the actions and methods for the tasks of the domain.

A planning problem is defined as a pair (Σ, ϕ_0), where Σ is a temporal planning domain and $\phi_0 = (\mathcal{A}, \mathcal{K}, \mathcal{T}, C)$ is a chronicle. This chronicle gives the tasks to perform, the additional goals to achieve, and the initially supported assertions in the current state of the world and future states that are expected to occur independently of the activities to be planned for. The pair $(\mathcal{T}, C)$ in ϕ_0 is required to be secure. Note that the planning problem ϕ_0 is defined in terms of tasks as well as goals. Planning proceeds by refinement of tasks as well as by generative search for goals.

Partial plans are also expressed as chronicles. A chronicle ϕ defines a solution plan when all its tasks have been refined and all its assertions are supported. At that point, ϕ contains all the actions initially in ϕ_0 plus those produced by the refinement of the tasks in ϕ_0, according to methods in Σ, and those possibly needed to support the assertions in ϕ_0 or required by the task refinements. It also contains the assertions and constraints in ϕ_0 plus those of the actions in ϕ, and the methods used in the task refinements, with their constraints and possible separations. More formally:

Definition 17.16. A chronicle ϕ is a valid solution plan of the temporal planning problem (Σ, ϕ_0) if and only if the following conditions hold:

(i) ϕ does not contain unrefined tasks;

(ii) all assertions in ϕ are causally supported, either by supported assertions initially in ϕ_0 or by assertions from methods and actions in the plan; and

(iii) the chronicle ϕ is secure. □

Condition *(i)* says that all tasks in ϕ_0 have been refined down into actions. Condition *(ii)* extends to temporal domains the notion of causal link seen in Section 3.4. Condition *(iii)* guarantees that the solution chronicle cannot have inconsistent instances, since is may have non-instantiated temporal and object variables, which will instantiated at execution time (see Chapter 18).

17.2.1 Temporal Planning Algorithm

A temporal planning algorithm proceeds by transforming the initial chronicle ϕ_0 with refinement methods and the addition of actions and separation constraints until the preceding three conditions are met. Let ϕ be the current chronicle in that transformation process; ϕ may contain three types of *flaws* with respect to the requirements of a valid plan in Definition 17.16:

- ϕ has unrefined tasks : violates condition *(i)*
- ϕ has unsupported assertions : violates condition *(ii)*, and
- ϕ has conflicting assertions : violates condition (*iii*).

Because ϕ is obtained by transforming ϕ_0, when ϕ does not contain unrefined tasks, then assertions not supported in ϕ are additional goals to be reached by adding in a generative way actions to the chronicle.

A flaw of one of the preceding three types is addressed by finding its *resolvers*, i.e., ways of resolving that flaw. The planning algorithm chooses a resolver nondeterministically and transforms the current chronicle accordingly. This is repeated until either the current chronicle is without flaws, that is, it is a valid solution or a flaw has no resolver, in which case the algorithm must backtrack to previous choices. This is specified in the pseudocode of TemPlan.

```
TemPlan($\phi, \Sigma$)
    while True do
        $Flaws \leftarrow$ set of flaws of $\phi$
        if $Flaws = \varnothing$ then return $\phi$
1       arbitrarily select $f \in Flaws$
2       $Resolvers \leftarrow$ set of resolvers of $f$
        if $Resolvers = \varnothing$ then return failure
3       nondeterministically choose $\rho \in Resolvers$
4       $\phi \leftarrow$ Transform($\phi, \rho$)
```

Algorithm 17.1. A schema of a chronicle temporal planner.

In TemPlan, Line 1 is a heuristic choice of the order in which the resolvers of a given flaw are searched. This choice affects the performance but not the completeness of the algorithm; Line 3 is a backtracking point in a deterministic implementation of TemPlan: all resolvers for a flaw may need to be tried to ensure completeness.

In addition to several design choices, the main technical issues for the implementation of this pseudocode into a temporal planner are the following:

- How to find the flaws in ϕ and their resolvers, and how to transform ϕ with a resolver ρ, that is, the Transform subroutine in TemPlan. This is discussed for the different types of flaws in Sections 17.2.2 to 17.2.4.
- How to organize and explore the search space efficiently. This is discussed in Section 17.2.5.
- How to check and maintain the consistency of the constraints in ϕ. This is discussed in Section 17.3.

An analogy with what was presented earlier can be helpful: resolving a unrefined task is like applying a method in HTN planning (Chapter 5); resolving an unsupported assertion is analogous to an open-goal flaw in Plan-Space Planning (Section 3.4); and resolving conflicting assertions is like resolving a threat in Plan-Space Planning.

17.2.2 Resolving Unrefined Tasks

An unrefined task is easy to detect in the current ϕ. A resolver for a flaw of that type is an applicable instance of a temporal refinement method for the task. An instance is obtained by renaming all variables in the method and instantiating some of these variables with the task parameters and with the variables and constraints of the current chronicle ϕ.

An instance m of a method is applicable to a chronicle ϕ when its task matches a task in ϕ and all the constraints of m are consistent with those of ϕ. Transforming $\phi = (\mathcal{A}, \mathcal{K}, \mathcal{T}, C)$ with such a resolver m consists of transforming ϕ is follows:

- replace the task in $\mathcal{A}$ with the subtasks and actions of m;
- add the assertions of m and those of the actions in m either to $\mathcal{K}$ if these assertions are causally supported, or else to $\mathcal{T}$;
- add to C the constraints of m and those of its actions.

An applicable instance of a method m may have assertions that are not causally supported by ϕ. For instance, in Example 17.12, the method m-bring is applicable for refining a task $\text{bring}(r, c, p)$ if m-bring has an instance such that the constraints $(\text{attached}(p', d'), \text{attached}(p, d), d \neq d', t_2 \leq t_1, t_3 \leq t_1)$ are consistent with those of current ϕ, given the current binding constraints of these variables. However, the assertion $[t_s, t_1]\,\text{freight}(r) = \text{empty}$ in that method may or may not be already supported by another assertion in ϕ. If it is not, then refining a task in ϕ with m-bring adds a unsupported assertion in the current chronicle.

17.2.3 Resolving Unsupported Assertions

Unsupported assertions in $\phi = (\mathcal{A}, \mathcal{K}, \mathcal{T}, C)$ are those initially in ϕ_0 plus those from the refinement of tasks and the insertion of actions. The three ways to support an assertion $\alpha \in \mathcal{T}$ and move it to $\mathcal{K}$ are the following:

- add constraints to C on object and temporal variables;
- add to $\mathcal{K}$ a persistence assertion; and
- add to $\mathcal{A}$ an action that brings an assertion supporting α, or a task with a method that brings such an assertion.

The last type of resolver corresponds to a generative planning mode. An unsupported assertion α is equivalent to a goal to be reached; α may be supported by either a primitive action or a method for a task. In both cases this introduces new tasks or new actions which do not result from the refinement of tasks. The use of an action as a resolver for supporting an unsupported assertion is similar to what we have seen for generative plan-space planners. Let us assume at this point that all actions in Σ can be freely used to augment a plan for supporting assertions, as well as through task refinement methods. We'll discuss this assumption in Section 17.2.7.

17.2.4 Resolving Conflicting Assertions

Flaws corresponding to conflicting assertions are more easily handled in an incremental way by maintaining ϕ as a secure chronicle and keeping track of what is needed for it to remain secure. The mechanisms here are a generalization of those used in Section 3.4 for handling threats in plan-space planning. There are, however, several substantial differences (see Exercise 17.8).

All assertions in ϕ_0 are required to be nonconflicting. Every transformation of ϕ by refinement, addition of persistence assertions or constraints, or addition of tasks or actions requires detecting and marking as flaws potential conflicts between newly added assertions and those of current ϕ. Resolvers for a potential conflict are sets of separation constraints consistent with the constraints in the current ϕ, as discussed in Section 17.1.1. The conflict is resolved by adding the chosen separation constraints to those of ϕ. One way of keeping the current ϕ secure is to detect and solve potential conflicts at every transformation step. However, other flaw selection strategies can be applied.

17.2.5 Search Space

The search space of TemPlan is a directed acyclic graph in which search states are chronicles. An edge (ϕ, ϕ') in this graph is such that $\phi' = \mathsf{Transform}(\phi, \rho)$, ρ being a resolver for some flaw in ϕ. The graph is acyclic because each edge augments the previous chronicle with additional constraints, actions, and/or assertions and there is no removal transformation. In general, however, the search space is not finite: it can grow indefinitely from the addition of new actions and tasks. It can be made finite by the specification of global constraints, such as the total duration of the plan.

Starting from ϕ_0, TemPlan explores a subtree of this complex search space. The problems for organizing and exploring this space are in many aspects similar to those of plans-space planning. Both follow the same approach of transforming a partial plan by finding flaws and repairing them. But their types of flaws are different. Flaws corresponding to unrefined tasks do no exist in PSP. The unsupported assertion flaws extend the open goal flaws of PSP to temporal domains. Similarly, conflicting assertions generalize what we referred to as threats in PSP.

Both TemPlan and PSP algorithms use a dynamic constraint-satisfaction approach in which new constraints and variables are repeatedly added during the search for a solution. The constraint-satisfaction problem (CSP) approach is very general and allows taking into account not only time and variable binding constraints, as in TemPlan, but also resource constraints, which are quite often part of planning problems. The *Meta-CSP framework*, which expresses the disjunctions of possible resolvers for flaws as (meta) constraints, can help formalize the integration of several types of constraints related to time and resources and possibly help in their resolution (see Section 17.4).

The basic heuristics for TemPlan are similar to those of PSP. These are basically variants of the *variable-ordering* and *value-ordering* heuristics of CSP. A heuristic analogous to variable-ordering chooses a flaw f that has the smallest number of resolvers (step *(i)* of TemPlan). For a heuristic analogous to value-ordering, the idea is to choose a resolver ρ that is the least constraining for the current chronicle ϕ. This notion is more difficult to assess; it leads to take into account differently resolvers that add constraints, assertions, or refinement methods, from those that add new tasks or actions. Adding new tasks and actions augments the size of the problem at hand and requires the use of more elaborate heuristics.

Advanced heuristics rely on elaborate extensions of domain transition graphs, reachability graphs and some of the techniques of Section 3.2. They can be integrated within various search strategies such as iterative deepening or A*-based search. These considerations are essential for designing an efficient implementation of TemPlan. Possible options for heuristics and search strategies are briefly discussed in Section 17.4.

TemPlan is sound when it is implemented with sound subroutines for finding flaws, resolvers and transforming chronicles. When a global constraint on the plan to find is set, such as the total duration of that plan or its maximum number of actions, then TemPlan is also complete, that is, at least one of its execution traces returns a solution plan, if there is one. These properties are conditioned on the soundness and completeness of the constraint handling procedures used in TemPlan, which are detailed in Section 17.3.

17.2.6 Illustration

Let us illustrate some of the steps of TemPlan on a detailed example.

Example 17.17. Consider the problem depicted in Figure 17.5 for the domain of Example 17.11 where two robots, r1 and r2, are servicing four docks, d1 to d4, connected with four roads, as illustrated. Starting from the initial state shown in the figure, the task is to bring the containers c1 to pile p3 and c2 to p4, with no constraint on the final robot locations. Hence, the initial chronicle ϕ_0 has two unrefined tasks and no unsupported assertion (see Exercise 17.5).

At the first recursion of TemPlan, there are two flaws in current ϕ: the unrefined tasks bring(r, c1, p3) and bring(r', c2, p4). Suppose the method m-bring is used to refine the first task into move, uncover, load, move and unload. At this point,

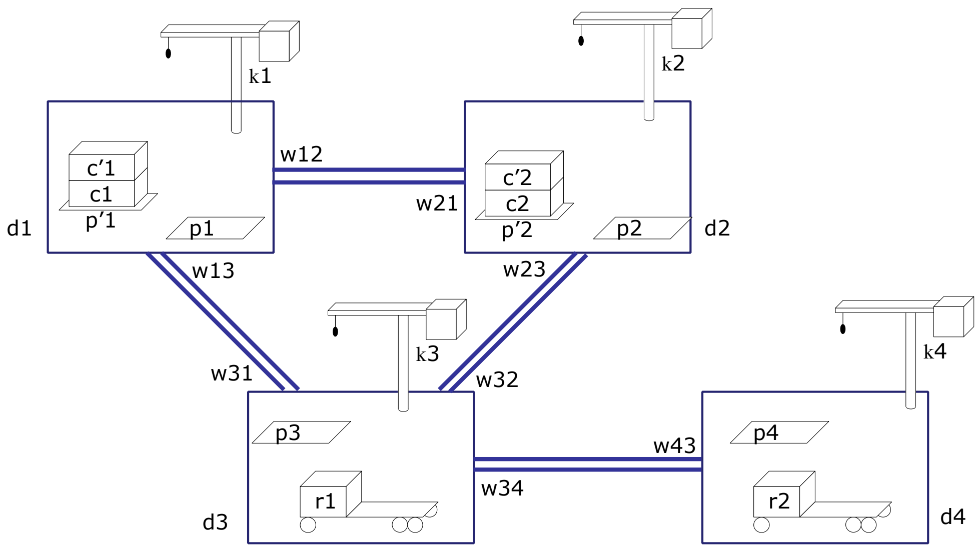


Figure 17.5. A planning problem involving two robots, r1 and r2, servicing four docks, d1 to d4; the task is to bring the containers c1 from pile p'1 to p3 and c2 from p'2 to p4.

c, p, p', d, d', k, k' can be instantiated,[2] respectively, to c1, p3, p'1, d3, d1, k1, k3; r is constrained to be in $\{$r1, r2$\}$ and the time points are constrained as depicted in Figure 17.3.

At the following recursion, there are six unrefined tasks in ϕ. Assume m-bring is similarly used to refine bring(r', c2, p4). Now the resulting chronicle contains ten unrefined tasks (two uncovers, loads and unloads, and four moves) as well as conflicting assertions related to the loc(r) and loc(r') assertions in the four load and unload tasks. Separation constraints are either $r \neq r'$ or precedence constraints such that the tasks are run sequentially.

If the former separation is chosen, a final solution plan would be, for example, to have r1 navigate to d2 while r2 navigates to d1. At the same time, k1 uncovers c1 while k2 uncovers c2. Two synchronizations then take place: before load(k2, r1, c2, p'2) and, concurrently, before load(k1, r2, c1, p'1) (as in Figure 17.3). These two concurrent actions are then followed by move(r1,d4) concurrently with move(r2,d3), and finally with the two unload actions. The details of the remaining steps for reaching a solution are covered in Exercise 17.6.

If we assume that navigation is constrained on the traversal of docks, that no dock can contain more than one robot at a time, then additional synchronizations will be required for the motion of the two robots (see Exercise 17.7). □

[2] An implementation may choose to instantiate partially the parameters of m-bring, but this would be more complicated to handle and may not pay off in efficiency.

17.2.7 Free Versus Task-Dependent Actions

The hybrid TemPlan integrates of a task refinement approach with a generative goal-oriented search approach. A chronicle $\phi_0 = (\mathcal{A}, \mathcal{K}, \mathcal{T}, \mathcal{C})$ specifies (in $\mathcal{A}$) the tasks to perform as well as (in $\mathcal{T}$) the goals to achieve in the form of temporal assertions. However, this flexible representation may limit the performance of the algorithm if the domain has few methods to depend on and relies significantly on generative search.

There is another issue regarding the use of actions to support assertions that relates to the specification style of a domain. An action a is specified as a collection of assertions and constraints; it is also a temporally qualified component of one or several methods. A method m using a may contain additional assertions and actions that can be needed for performing a. A domain may or may not allow a to be freely used in a generative approach, independently of methods that refine a task into several actions, including a.

These considerations motivate a distinction between *free* and *task-dependent* actions. An action is free if it can be used alone for supporting assertions. An action is task-dependent if it can be used only as part of a refinement method in generative planning. Such a property is a matter of design and specification style of the planning domain.

Example 17.18. The designer of the domain in Example 18.4 may consider that the actions unload, load, stack, and unstack are free. These actions can be performed whenever their specified conditions are met; they can be inserted in a plan when their assertions are needed to support unsupported assertions. However, the actions leave and enter can be specified as being task-dependent; they should necessarily appear as the result of a decomposition of a move task. Here, the designer does not foresee any reason to perform actions leave or enter except within move tasks that require leaving or entering a place. □

The use of a task-dependent action branches over the choice of which task to use if the same action appears in the decomposition of several tasks. It introduces an unrefined task flaw, which branches over several methods for its decomposition. Note that if all actions in a domain are free, then the refinement in TemPlan is limited to the tasks in the initial chronicle. But if all actions are task-dependent, then refinement will be needed for every unsupported assertion that cannot be supported by constraints and persistence assertions.

17.3 Constraint Management

At each recursion of TemPlan, we have to find resolvers for current flaws and transform the current chronicle ϕ by refinement and insertion of assertions, constraints, actions, and tasks. Each transformation must keep the set C of constraints in ϕ consistent; it must detect conflicts in the set of assertions in ϕ and find separation constraints consistent with C. The steps *(ii)* and *(iv)* of TemPlan require checking the consistency of the constraints in C.

There are two types of constraints in C: temporal constraints and object constraints. Let us assume that these two types of constraints are *decoupled*, that is, there is no constraint that restricts the value of a time point as a function of object variables, or vice versa. For example, we introduced constant parameters δ_i in Example 17.11; there would be a coupling if these delays where not constant but functions of which robot r is doing the leave or which crane the unload actions. With this simplifying assumption, C is consistent if and only if the object constraints and the temporal constraints are consistent. Constraint checking relies on two independent constraint managers for the two types of constraints, discussed next.

17.3.1 Consistency of Object Constraints

A temporal planner must check and maintain the consistency of unary and binary constraints on object variables that come from binding and separation constraints and from rigid relations. This corresponds to maintaining a general CSP over finite domains, the consistency checking of which is an NP-complete problem. Restrictions on the representation that can give a tractable CSP are not practical; even inequality constraints, such as $x \neq y$ in a separation constraint, make consistency checking NP-complete.

Filtering techniques, such as incremental arc or path consistency, are not complete, but they are efficient and offer a reasonable trade-off for testing the consistency of object constraint networks. Indeed, if TemPlan progresses with an inconsistent set of object constraints, it will later detect that some variables do not have consistent instantiations; it will have to backtrack. Incomplete consistency checking in each search node does not reduce the completeness of the algorithm, it just may not prune earlier some nodes in the search tree. There is trade-off between *(i)* running a complete but costly consistency checking at each search node for early pruning, and *(ii)* doing a fast incremental constraint filtering, postponing a complete variable instantiation checking to the end of the search, which may require further backtracking.[3]

A good principle for balancing this trade-off is to perform low complexity procedures at each search node, and to keep more complex ones as part of the search strategy. In that sense, filtering techniques efficiently remove many inconsistencies and reduce the search space at a low cost. They may be used jointly with complete algorithms, such as forward-checking at regular stages of the search. Such a complete consistency check has to be performed on the free variables remaining in the final plan. Other trade-offs, such as choosing flaws that lead to instantiate object variables, are also relevant for reducing the complexity of maintaining variable binding constraints.

17.3.2 Consistency of Temporal Constraints

Simple Temporal Networks (STNs) provide a convenient framework for handling temporal constraints. An STN is a pair $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of temporal

[3] As an analogy with plan-space planning (Section 3.4), PSP checks the consistency of the relation $\prec$, but ignores other constraints that are less easily checked. For example, suppose PSP constrains $d \neq \mathsf{d1}, d \neq \mathsf{d2}, d \neq \mathsf{d3}$, if $Range(d) = \{\mathsf{d1}, \mathsf{d2}, \mathsf{d3}\}$, PSP will discover the inconsistency only through search.

variables $\mathcal{V} = \{t_1, t_2, \ldots, t_n\}$, and $\mathcal{E}$ is a set of binary constraints of the form:

$$a_{ij} \leq t_j - t_i \leq b_{ij}, \text{ where } a_{ij} \text{ and } b_{ij} \text{ are integers.}$$

We use the notation $r_{ij} = [a_{ij}, b_{ij}]$ for $t_j - t_i \in [a_{ij}, b_{ij}]$. Note that r_{ij} entails $r_{j,i} = [-b_{ij}, -a_{ij}]$. To represent unary constraints (i.e., constraints on one variable rather than two), we use a fixed time point $t_0 = 0$. The constraint $a \leq t_j \leq b$ is denoted $r_{0j} = [a, b]$.

A solution to an STN $(\mathcal{V}, \mathcal{E})$ gives an integer value to each variable in $\mathcal{V}$. The STN is *consistent* if it has a solution that meets all the constraints in $\mathcal{E}$. It is *minimal* if every value in each interval r_{ij} belongs to a solution (see exercises 17.10 to 17.14).

TemPlan proceeds by transforming a chronicle $\phi = (\mathcal{A}, \mathcal{K}, \mathcal{T}, \mathcal{C})$ such as to meet the conditions of a solution plan. These transformations add in $\mathcal{C}$ constraints of methods for refining tasks, constraints for supporting assertions, and separation constraints for conflicting assertions. Each transformation should keep $\mathcal{C}$ consistent. The set of temporal constraints in $\mathcal{C}$ is an STN, which evolves by adding new variables and constraints while staying consistent. TemPlan has to check incrementally that this STN remains consistent when more variables and contraints are added to it. This is more easily done when the network it is also maintained minimal, as explained next.

Two operations are essential for checking the consistency of $\mathcal{E}$:

- *composition*: $r_{ik} \bullet r_{kj} = [a_{ik} + a_{kj}, b_{ik} + b_{kj}]$, which corresponds to the transitive sum of the two constraints from i to j through k:
  $a_{ik} \leq t_k - t_i \leq b_{ik}$ and $a_{kj} \leq t_j - t_k \leq b_{kj}$;
- *intersection*: $r_{ij} \cap r'_{ij} = [\max\{a_{ij}, a'_{ij}\}, \min\{b_{ij}, b'_{ij}\}]$, which is the conjunction of two constraints on (t_i, t_j): $a_{ij} \leq t_j - t_i \leq b_{ij}$ and $a'_{ij} \leq t_j - t_i \leq b'_{ij}$.

Three constraints r_{ik}, r_{kj}, and r_{ij} are consistent when $r_{ij} \cap (r_{ik} \bullet r_{kj}) \neq \emptyset$.

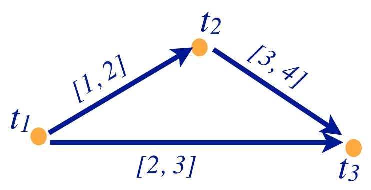


Figure 17.6. A simple temporal network.

Example 17.19. Consider the network in Figure 17.6 where vertices are time points and edges are labelled with temporal constraints: $r_{12} = [1, 2], r_{2,3} = [3, 4]$ and $r_{13} = [2, 3]$. Then

$$r_{12} \bullet r_{23} = [1 + 3, 2 + 4] = [4, 6],$$
$$\text{so} \quad r_{13} \cap (r_{12} \bullet r_{23}) = [2, 3] \cap [4, 6] = \emptyset,$$

so the network is inconsistent.

Next, suppose that $r_{12} = [1, 2]$ and $r_{2,3} = [3, 4]$ as above, but $r_{13} = [2, 5]$. Then

$$r_{13} \cap (r_{12} \bullet r_{23}) = [2, 5] \cap [4, 6] = [4, 5],$$

so we can make r_{13} minimal by assigning $r_{13} \leftarrow [4, 5]$. From this, we can compute

$$r_{12} \cap (r_{13} \bullet r_{32}) = [4 - 4, 5 - 3] \cap [1, 2] = [1, 2],$$
$$r_{23} \cap (r_{21} \bullet r_{13}) = [4 - 2, 5 - 1] \cap [3, 4] = [3, 4],$$

so r_{12} and r_{23} are already minimal. Thus the entire network is minimal. □

The path-consistency algorithm PC (Algorithm 17.2) tests all triples of variables in $\mathcal{V}$ with a *transitive update* operation: $r_{ij} \leftarrow r_{ij} \cap (r_{ik} \bullet r_{kj})$. If a pair (t_i, t_j) is not constrained, then we take $r_{ij} = (-\infty, +\infty)$; in that sense, an STN corresponds implicitly to a complete graph.

```
PC($\mathcal{V}, \mathcal{E}$)
    for $k = 1, \dots, n$ do
        foreach $i \neq k$ and $j \neq k$ such that $1 \leq i < j \leq n$ do
            $r_{ij} \leftarrow r_{ij} \cap [r_{ik} \bullet r_{kj}]$
            if $r_{ij} = \varnothing$ then return inconsistent
```

Algorithm 17.2. PC path consistency algorithm for simple constraint networks

PC is complete and returns the minimal network. Its complexity is $O(n^3)$. It is easily transformed into an incremental version. Assume that the current network $(\mathcal{V}, \mathcal{E})$ is consistent and minimal; a new constraint r'_{ij} is inconsistent with $(\mathcal{V}, \mathcal{E})$ if and only if $r_{ij} \cap r'_{ij} = \varnothing$. Furthermore, when $r_{ij} \subseteq r'_{ij}$ the new constraint does not change the minimal network $(\mathcal{V}, \mathcal{E})$. Otherwise r_{ij} is updated as $r_{ij} \cap r'_{ij}$ and propagated over all constraints r_{ik} and r_{kj} with the transitive update operation; any change is subsequently propagated. Incremental path consistency is in $O(1)$ for consistency checking and in $O(n^2)$ for updating a minimal network.

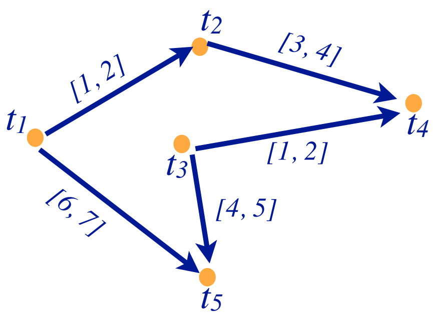


Figure 17.7. A consistent STN.

Example 17.20. Let us give the network in Figure 17.7 as input to PC. The first iteration for $k = 1$ with $2 \leq i < j \leq 5$ does not change the constraints $r_{23}, r_{24}, r_{25}, r_{34}, r_{35}$; it updates r_{25} as follows: $r_{25} \leftarrow r_{25} \cap [r_{21} \bullet r_{15}] = (-\infty, +\infty) \cap [-2, -1] \bullet [6, 7] = [4, 6]$. The remaining iterations confirm that this network is consistent and minimal (see Exercise 17.10). □

The consistency of STNs can also be maintained with the Floyd-Warshall all-pairs minimal distance algorithm. Here, a network $(\mathcal{V},\mathcal{E})$ is transformed into a distance graph, the vertices of which are again the time points in $\mathcal{V}$. Each constraint $r_{ij} = [a_{ij}, b_{ij}]$ of the network defines two edges in the graph: *(i)* an edge from t_i to t_j labelled with a distance b_{ij}, and *(ii)* an edge from t_j to t_i labelled with a distance $-a_{ij}$. The original network is consistent if and only if there is no negative cycle in this distance graph. The Floyd-Warshall algorithm checks the consistency and computes minimal distances between all pairs of vertices in the graph in $O(n^3)$ time. An incremental version of this algorithm has been devised for planning. Another alternative is the Bellman-Ford algorithm which computes the single source distances in the distance graph. It can be used for consistency checking with a complexity in O($n \times m$), where n is the number of vertices and m the number of edges of the distance graph. The graph is kept sparse ($m < n^2$), but the algorithm does not maintain a minimal network. There is also an incremental version of this algorithm.

When TemPlan returns a valid solution plan ϕ, the temporal variables in ϕ are consistently constrained and non-instantiated. They are instantiated at acting time. Some of these variables can be freely chosen by the actor within their consistency intervals, e.g., when to trigger an action. But other variables will be set by the environment and can only be observed, e.g., when an action finishes. These *uncontrollable* variables raise additional consistency issues at acting time, that will be addressed in the next chapter.

17.4 Discussion and Bibliographic Notes

Temporal Representation and Reasoning Temporal models are widely used in AI well beyond planning. Numerous studies are devoted to knowledge representations and reasoning techniques for handling time, in particular, for dealing with change, events, actions, and causality; see e.g., [29, 773, 1012, 1011, 977], and the handbook of [364].

Most of the work cited above relies on a state-oriented view based on various temporal logics. The timeline approach, developed in this chapter, decomposes a reasoning task into a specialized solver say a planner and a temporal reasoner, that maintains, through queries and updates, a consistent network of temporal references. In addition to planning, this approach is used in other applications, such as temporal databases [184], monitoring [925], diagnosis [183, 689], multi-media document management [341, 10], video interpretation [1144], and process supervision [308, 307].

Temporal networks can use as primitives either time points or intervals. They manage either qualitative or quantitative constraints. A synthetic introduction to temporal networks can be found in [410, chapter 13] or [93].

Qualitative approaches to temporal reasoning were introduced by [31] with a specific algebra over intervals and a path consistency filtering algorithm. The time point algebra is introduced in [1140]. Tractable subclasses of the interval or the time point algebra have been studied [1141, 782, 835, 309]. Other authors studied representations integrating time points and intervals and their tractable subclasses [716, 437, 559].

Quantitative approaches to handling time relied initially on linear equations and linear programming techniques, e.g., [750]. Temporal constraint satisfaction problems and the STN tractable subclass, used in this chapter, were introduced by [286]. Several improvements have been proposed, e.g., for the incremental management of STNs [212, 904]. Various extensions to STNs have been studied, with preferences [601] or specific constraints in [644].

Constraints in planning play an important role. Authors have sought ways to efficiently structure them, e.g., with meta-CSPs. A meta-CSP is a CSP above lower-level CSPs. Its meta-variables are the lower level constraints; their values are alternative ways to combine consistently these constraints. For example, with disjunctive temporal constraints the values correspond to possible disjuncts. Meta-CSPs have been used in different settings, e.g., the management of preferences in temporal reasoning [802, 801, 93]. They have been applied to temporal planning by several authors, e.g., [401, 956, 446]. The approach appears elegant for handling temporal and other constraints on different kind of resources [752], but is not easily scalable.

Temporal Planning. There is a long history of research in temporal planning. Numerous temporal planners have been proposed, starting from early temporal HTN planners such as Deviser [1128], SIPE [1171], FORBIN [283] or O-PLAN [265]. These planners integrate temporal extensions to HTN representations and algorithms.

The state-oriented view in temporal planning extends the classical model of instantaneous precondition-effect transitions with *durative actions*. The basic model considers a start point and a duration. It requires preconditions to hold at the start and effects at the end of an action. This is illustrated in TGP [1033] or in TP4 [477]. Extensions of this model with conditions that prevail over the duration of the action, (as in the model of [978]) have been proposed, for example, in the SAPA planner [299] or in PDDL2.1 [370] and other temporal extensions of PDDL [481, Chapter 5]. Several planners rely on the latter representation, such as HS [477], TPSYS [388] and CRIKEY [248].

A few planners using the durative action model adopt the plan-space approach, notably Zeno [882] which relies on linear programming techniques, or VHPOP [1214] which uses STN algorithms. Some planners pursue the HTN approach, as the earlier planners mentioned above, or more recently SHOP2 [833] or Siadex [209]. The latter is also hybrid.

Many durative actions planners rely on state-based search techniques. A few are based on temporal logic approaches. Among these are TALplanner [300, 661], a model-checking based planner [321], or a SAT-based planner [528]. Significant effort has been invested in generalizing classical state-space planning heuristics to the durative action case. The action compression technique, which basically abstract the durative transition to an instantaneous one for the purpose of computing a heuristic, is quite popular, for example in [398] or [336]. Various temporal extensions of the relaxed planning graph technique (Section 3.2.3), as in Metric RPG [506], have been proposed, e.g., [477, 733, 248] and [475]. Sampling over a duration interval with action compression has also been investigated [604].

A few durative action planners handle hybrid discrete-continuous change. Some planners address continuous effects through repeated discretization, for example, UPMurphy [885]. Linear programming techniques, when the continuous dynamics is assumed to be linear, have been used since ZENO [882] by several planners. A recent and elaborate example is COLIN [249]. The Kongming planner [705] relies on domain specific dynamic models.

While the durative action model led to quite performant planners, it usually has a weak notion of concurrency that basically requires independence between concurrent actions. Interfering effects, as discussed in Example 17.15, can be addressed by a few of the above mentioned planners, for example, notably COLIN [249]. Alternatively, interfering effects can be addressed with the time-oriented view.

Planning along the time-oriented view was introduced [33] in a planner based on the interval algebra and plan-space search [32, 30]. The Time-Map Manager of [282] led to the development of a few planners [283, 147] and several original ideas related to temporal databases and temporal planning operators.

Planning with chronicles was introduced in IxTeT [408, 406]. The IxTeT kernel is an efficient manager of time point constraints [407]. IxTeT handles concurrent interfering actions, exogenous events and goals situated in time. It uses distance-based heuristics [381] integrated to abstraction techniques in plan-space planning. Its performance and scalability were improved by several other timeline oriented planners using similar representations. These are notably ParcPlan [323, 713], ASPEN [926], PS [558], IDEA [824], EUROPA [372], APSI of [375], and T-REX [920, 929, 931].[4] Elaborate heuristics, generalizing the reachability and dependency graphs of state-space planning, have been designed for these representations, for example, by [121]. A few of the mentioned planners have been deployed in demanding applications, for example, for controlling autonomous space systems and underwater vehicles.

An interesting development has been brought by the Action Notation Modeling Language (ANML) [1037]. ANML is timeline-based expressive representation, as presented in this chapter, which allows to specify hybrid temporal planning problem with tasks and goals. FAPE [316, 143] is a hybrid planner and acting system based on ANML. Note also the representation called HDDL2.1 [881] which extends HDDL with temporal constructs; there HTN methods correspond to temporal task networks specifying temporal constraints for task decompositions.

Refinement methods reduce the search complexity by providing domain-specific knowledge, but they do not palliate the need of good heuristics. Some temporal logic based planners, like TALplan, rely on control rules. Most state-based temporal planners referred to earlier exploit the techniques of Section 3.2. The use of classical planning heuristics has been an important motivation for the state-oriented view of temporal planning. These techniques have been extended to plan-space planning

[4]The terminology of "*Timeline planning*" often refers to a particular case of chronicle planning where temporal constraints are restricted to qualitative temporal interval algebra.

(e.g., in RePop [849] and VHPOP [1214]) and further developed for timeline based planners. There is notably the mutual exclusion technique [118] and the dependency graph approach [121]. Dependency graphs record relationship between possible activities in a domain. They are based on activity transition graphs [119, 120], which are a direct extension of the domain transition graphs of state variables [490]. These techniques have been successfully demonstrated in EUROPA2 [120, 121].

Temporal planning has naturally been associated with resources handling capabilities. Several of the planners mentioned above integrate planning and scheduling functions, in particular with constraint-based techniques, which where introduced early in IxTeT by [665]. Algorithmic issues for the integration of resource scheduling and optimization in planning attracted numerous contributions [1036, 213, 664, 1129]. A global overview of scheduling and resource planning is proposed in [81].

17.5 Exercises

17.1. Here are temporal versions of the blocks-world actions from Exercise 2.4:

pickup(x)
 assertions: $[t_s, t_e]\text{holding} : (\text{nil}, x)$
 $[t_s, t_e]\text{pos}(x) : (\text{table}, \text{nil})$
 $[t_s, t_e]\text{clear}(x) = \text{T}$
 constraints: $t_s < t_e$

unstack(x, y)
 assertions: $[t_s, t_e]\text{holding} : (\text{nil}, x)$
 $[t_s, t_e]\text{pos}(x) : (y, \text{nil})$
 $[t_s, t_e]\text{clear}(x) = \text{T}$
 $[t_s, t_e]\text{clear}(y) : (\text{F}, \text{T})$
 constraints: $t_s < t_e$

putdown(x)
 assertions: $[t_s, t_e]\text{holding} : (x, \text{nil})$
 $[t_s, t_e]\text{pos}(x) : (\text{nil}, \text{table})$
 constraints: $t_s < t_e$

stack(x, y)
 assertions: $[t_s, t_e]\text{holding} : (x, \text{nil})$
 $[t_s, t_e]\text{pos}(x) : (\text{nil}, y)$
 $[t_s, t_e]\text{clear}(y) : (\text{T}, \text{F})$
 constraints: $t_s < t_e$

Each of the following chronicles has exactly one flaw. For each of them, tell:

(a) What the flaw is.
(b) What resolver to use. If there is more than one candidate, choose the one that leaves the fewest flaws in the chronicle.
(c) What changes the resolver will make to the chronicle.
(d) What flaws the chronicle will have after the resolver has been applied.

ϕ_0:
 supported: $[t_0]\text{pos}(\text{a}) = \text{b}$
 $[t_0]\text{pos}(\text{b}) = \text{table}$
 $[t_0]\text{clear}(\text{a}) = \text{T}$
 $[t_0]\text{clear}(\text{b}) = \text{F}$
 $[t_0]\text{holding} = \text{nil}$
 assertions: $[t_1]\text{pos}(\text{a}) = \text{b}$
 constraints: $t_0 < t_1$

ϕ_1:
 supported: $[t_0]\text{pos}(\text{a}) = \text{b}$
 $[t_0]\text{pos}(\text{b}) = \text{table}$
 $[t_0]\text{clear}(\text{a}) = \text{T}$
 $[t_0]\text{clear}(\text{b}) = \text{F}$
 $[t_0]\text{holding} = \text{nil}$
 assertions: $[t_1]\text{pos}(\text{a}) = \text{table}$
 constraints: $t_0 < t_1$

17.2. Specify the actions stack, unstack and navigate of Example 17.11. For the latter, assume that navigation between connected waypoints is unconstrained.

17.3. For the domain in Example 17.12, define methods for the tasks load and unload. For the task bring, define additional methods to cover the following cases:

- the destination pile is at the same dock as the source pile,
- the robot r is already loaded with container c,
- container c is already in its destination pile.

Similarly, define other methods for the task move to cover the cases where the robot starts from a waypoint or when it is already at destination, and another method for the task uncover when the container c is at the top of pile p.

17.4. Augment the domain of Example 17.12 by considering that a pile p can be attached to a ship and that a crane k on a dock d can unstack containers from a pile p only when the corresponding ship is docked at d.

17.5. Specify the initial chronicle ϕ_0 for the problem of Example 17.17 and Figure 17.5.

17.6. In Example 17.17, develop the steps of TemPlan until reaching a solution to the planning problem.

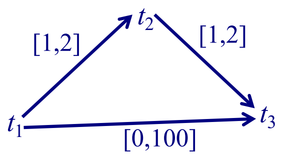
17.7. For the domain in Example 17.12, redefine navigate as a task which refines into the traversal of roads and the crossing to docks. The navigation between two roads adjacent to a dock d requires crossing d which should not be occupied during the crossing interval. For example, in Figure 17.5 the navigation from d4 to d1 requires the traversal of d3 which should be empty when the robot gets there. Analyze the conflicting assertions that result from this modification in the first few steps of TemPlan for Example 17.17 and find resolvers for the corresponding flaws.

17.8. Analyse the commonalities and differences between the notion of threats in Section 3.4 and that of conflicting assertions. Notice that the former relate actions while the latter are with respect to assertions. Since a threat is a menace to a causal link, can there be conflicting assertions without a causal support? If the answer is affirmative, give an example.

17.9. In Example 17.17, implement the modification introduced in Exercise 17.5: consider that piles p'1 and p'2 are not fixed in their respective docks but attached to two ships that will be docked respectively to d1 and d2 at two future intervals of time $[t_1, t_1+\delta_1]$ and $[t_2, t_2+\delta_2]$. How is modified the solution found in Exercise 17.6 when these two intervals do not overlap. What happens when $[t_1, t_1 + \delta_1]$ and $[t_2, t_2 + \delta_2]$ are overlapping?

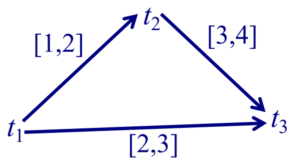
17.10. Run the algorithm PC on the networks in Figure 17.7. Show that it adds the constraints $r_{1,3} = [1, 3]$, $r_{24} = [1, 2]$ and $r_{45} = [2, 3]$.

17.11. Show that the set of solutions of the following network include $(t_2 - t_1, t_3 - t_2, t_3 - t_1) \in \{(1, 1, 2), (1, 2, 3), (2, 1, 3), (2, 2, 4)\}$.

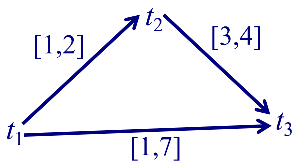


17.12. Find the minimal network corresponding to the previous example

17.13. Is this network consistent?



17.14. Is this network minimal?



17.15. Specify and implement an incremental version of the PC algorithm; use it to analyze how the network in Figure 17.7 evolves when are added to it successively $t_6, r_{36} = [5, 8]\,, r_{56} = [2, 5]$ then $t_7, r_{47} = [3, 6]\,, r_{67} = [1, 7]$.

17.16. Run algorithm PC on the networks in Figure 18.1 and Figure 18.2 and compute all the implicit constraints entailed from those in the networks; show that both networks are minimal.

17.17. Prove that the minimal network in Figure 18.3 is such that $[b - v, a - u] \subseteq [p, q]$.

17.18. Consider the minimal network in Figure 18.3 for the case where $u \geq 0$ and $[b - v, a - u] = \varnothing$. Prove that this network is not dynamically controllable.

17.19. Consider the temporal network associated with the solution of Exercise 17.6: under what condition is it dynamically controllable?

17.20. For all the actions in Example 17.11, define atemporal acting refinement methods similar to the two given in Example 18.8.

17.21. Run algorithm Dispatch for the solution plan found in Exercise 17.6 assuming that robot r1 is much faster than r2.

18 Acting with Temporal Controllability

Consider an actor that has to follow a temporal plan $\phi = (\mathcal{A}, \mathcal{S}_\mathcal{T}, \mathcal{T}, C)$ associated with a consistent temporal network C. The time points in C are only constrained, they are not precisely set in advance by a planner. They are instantiated at acting time and depend on how execution is pursued by the platform given temporal uncertainties.

The actor is faced with a controllability issue: not all time points in C are under its control. It chooses freely, within the allowed bounds, when to trigger an action. The action's duration will also be within the bounds in C (assuming a correct model). But in general, the actor cannot choose when precisely an action finishes. It can only observe the value of that instant.[1] This observed value further contrains the network C and may interfere with its consistency. In addition to durations, the time occurrence of an expected contingent event is not controllable. The actor needs a *dynamically controllable network*, that remains consistent for *all* possible values, within the modeled bounds, of uncontrollable time points. If C meets this property, the actor uses a *dispatching* algorithm to trigger the controllable time points such as to keep the rest of the plan feasible.

Given a dynamically controllable plan ϕ, the actor may execute it as is, or use refinement methods to further refine the actions in ϕ opportunistically at acting time. If no plan is given or if it fails, the actor may use temporal refinement methods to act reactively without a plan (as in Chapter 14), or use them with Monte Carlo rollouts to perform guiding lookahead (as in Chapter 15).

This chapter addresses the issues of dynamic controllability (Section 18.1), dispatching (18.2), execution and refinement of a temporal plan (18.3), acting with a reactive temporal refinement engine (18.4), planning with Monte Carlo rollouts (18.5), and integrating planning and acting (18.6).

18.1 Controllable Temporal Networks

Let t_s and t_e be the start and end time points of an action a in a plan ϕ. The actor can trigger a at any time according to the constraints on t_s. The precise triggering moment of a is under its control. However, the moment at which the action terminates, and the other intermediate instants while the action is taking place, are generally not under its control. These time points are observable: the execution platform reports when the action terminates and when the intermediate time points in the action model are reached. But they are uncontrollable, and said to be *contingent*.

Let us see how to deal with contingent points. The path consistency procedure PC (Algorithm 17.2) checks a temporal network by *squeezing* intervals until reaching a

[1] It may interrupt the action, but generally this is a failure case.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

minimal network: it reduces intervals with the update operation $r_{ij} \leftarrow r_{ij} \cap (r_{ik} \bullet r_{kj})$. This does not work for contingent points. Consider an action a in $[t_s, t_e]$ has a constraint: $l \leq t_s - t \leq u$. This requirement on t_s can be met by choosing freely the starting point in the range $[l, u]$ after observing t. If required for meeting other constraints, this interval can be *squeezed* into any other nonempty interval $[l', u'] \subseteq [l, u]$. However, a constraint on the end point of action a such as $l \leq t_e - t_s \leq u$, has a different meaning; it says that the duration of the interval $[t_s, t_e]$ is a contingent value in the range $[l, u]$. This duration will be *observed* once a terminates. It will range in the uncertainty interval $[l, u]$. The actor has no freedom for the choice of t_e. A duration is seldom controllable, while PC works as if it is.

These considerations are not specific to action durations. They hold for any contingent time points and constraints. They apply in particular to contingent expected events that can be specified in the initial chronicle (as in Example 17.14). The time distance between an absolute reference point and an expected event is an uncontrollable duration, similar to an action duration.

Example 18.1. Consider the robot of Example 17.12 that has to achieve a task, denoted bring&move, that will take it to dock1. Concurrently, a crane at dock1 has to uncover a container that will be loaded on the robot. The duration of bring&move from t_1 to t_2 is specified in the model of the task to be in $[30, 50]$ time units; task uncover from t_2 to t_3 takes 5 to 10 time units. Further, the initial chronicle requires the two tasks to be synchronized such that neither one lags after the other by more than 5 time units, that is: $-5 \leq t_4 - t_2 \leq 5$ (see Figure 18.1(a)).

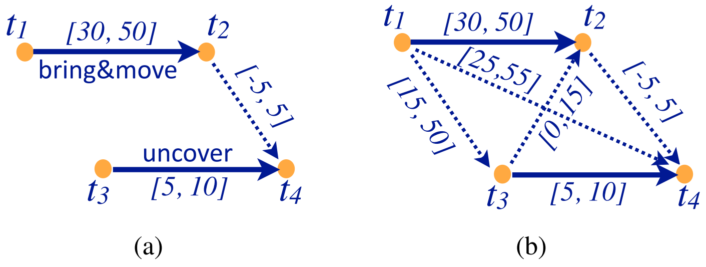


Figure 18.1. An uncontrollable network. Tasks are depicted as plain arrows, synchronization constraints as dashed arrows.

A direct application of PC to the network in Figure 18.1(a) shows that this network is consistent; it returns the minimal network in Figure 18.1(b) (see Exercise 17.16). Suppose this network is used by an actor who only controls the triggering of the two tasks, that is, t_1 and t_3. It is clear that t_1 should precede t_3 because $[t_1, t_3] \subseteq [15, 50]$. Suppose the first task is triggered at time $t_1 = 0$. When should the second task be triggered such as to meet the synchronization constraint between t_2 and t_4?

Let d and d' be the respective durations of the two tasks. The synchronization constraint says $-5 \leq t_4 - t_2 \leq 5$, that is, $-5 \leq t_3 + d' - d \leq 5$. The choice of t_3 should satisfy the constraints $d - d' - 5 \leq t_3$ and $t_3 \leq d - d' + 5$ for *all* possible values of d and d' in their respective intervals, which is not feasible (e.g., $d = 50, d' = 5$ entails $40 \leq t_3$, while $d = 30, d' = 10$ requires $t_3 \leq 25$).

Earlier, we said that PC finds this STN consistent. But PC assumes full control over *every* variable, which is not the case here. One can easily check that there is no problem in meeting all the constraints if one can freely choose d and d' in their intervals, for example, $d = 30, d' = 10$ leaves $t_3 \in [15, 25]$.

The actor does not control the end points of actions. However, it can observe them, and may devise a *conditional* strategy based on what it observes. For example, it may start uncover at most 40 units after t_1, or earlier if bring&move finishes before t_1. In this particular example, such a strategy does not work, but if the actor can observe an intermediate time point between t_1 and t_2, this may make its synchronization problem controllable, as explained next. □

Simple Temporal constraint Networks with Uncertainty. STNUs extend STNs by partitioning time points and constraints into controllable ones and contingent ones.

Definition 18.2. An STNU is a tuple $(\mathcal{V}, \tilde{\mathcal{V}}, \mathcal{E}, \tilde{\mathcal{E}})$, where $\mathcal{V}$ and $\tilde{\mathcal{V}}$ are disjoint sets of time points, and $\mathcal{E}$ and $\tilde{\mathcal{E}}$ are disjoint sets of binary constraints on time points. $\mathcal{V}$ and $\mathcal{E}$ are said to be *controllable*; $\tilde{\mathcal{V}}$ and $\tilde{\mathcal{E}}$ are said to be *contingent*. If $[l, u]$, with $0 < l < u < \infty$, is a contingent constraint in $\tilde{\mathcal{E}}$ on the time points $[t_s, t_e]$, then t_e is a contingent point in $\tilde{\mathcal{V}}$. □

The intuition is that elements in $\tilde{\mathcal{V}}$ denote the *ending* time points of actions, while contingent constraints in $\tilde{\mathcal{E}}$ model the positive nonzero durations of actions, predicted with uncertainty. If $[t_s, t_e] \subseteq [l, u]$ is a contingent constraint, then the actual duration $t_e - t_s$ is also a contingent variable whose value will be observed within $[l, u]$, once the corresponding action terminates. The actor *controls* t_s; it assigns a value to it. It only observes t_e, knowing in advance that it will be within the bounds set for the contingent constraint on $t_e - t_s$. An STNU cannot have a contingent variable t_e that is the end point of two contingent constraints.

The controllability issue is to make sure (at planning time) that there exist values for the controllable variables such that for *any* observed value of the contingent variables within their bounds, the contraints will be satisfied. One can view controllable variables as being existentially quantified, while contingent ones are universally quantified. However, the actor does not need to commit to values for all of its controllable variables *before* starting to act. It can choose a value for a controllable variable only when the value is needed at acting time. It can make this choice as a function of the observed values of *past* contingent variables.

Definition 18.3. A *dynamic execution strategy* for an STNU $(\mathcal{V}, \tilde{\mathcal{V}}, \mathcal{E}, \tilde{\mathcal{E}})$ is a procedure for assigning, at acting time, values to controllable variables $t \in \mathcal{V}$ consistent with $\mathcal{E}$ such that all the constraints in $\mathcal{E}$ related to t are satisfied, given that the values of all contingent variables in $\tilde{\mathcal{V}}$ preceding t are known and fit the constraints in $\tilde{\mathcal{E}}$. An STNU is *dynamically controllable* if it has a dynamic execution strategy. □

The actor's interaction with its environment can be modeled game-theoretically as follows. For $i = 0, 1, 2, \ldots,$

1. The actor chooses a set of unassigned controllable variables $\mathcal{V}_i \subseteq \mathcal{V}$ (that is, actions to start at time i) that all can be assigned the value i without violating any constraints in $\mathcal{E}$. These variables are set to i.
2. The *world* chooses a set of unassigned contingent variables $\tilde{\mathcal{V}}_i \subseteq \tilde{\mathcal{V}}$ (that is, events observed at time i) that can have the value i without violating any constraints in $\tilde{\mathcal{E}}$. The actor observes that these variables are set to i.
3. The process ends with failure if any of the constraints in $\mathcal{E} \cup \tilde{\mathcal{E}}$ are violated. These might include violations that neither $\mathcal{V}_i$ nor $\tilde{\mathcal{V}}_i$ caused individually.
4. The process ends with success if all variables in $\mathcal{V} \cup \tilde{\mathcal{V}}$ have values and no constraints are violated.

In this model, the actor's dynamic execution strategy is the algorithm that it uses to choose each set $\tilde{\mathcal{V}}_i$ in step 1. The STNU $(\mathcal{V}, \mathcal{E}, \tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ is dynamically controllable if there exists a dynamic execution strategy such that the game ends with success regardless of the world's choices in step 2.

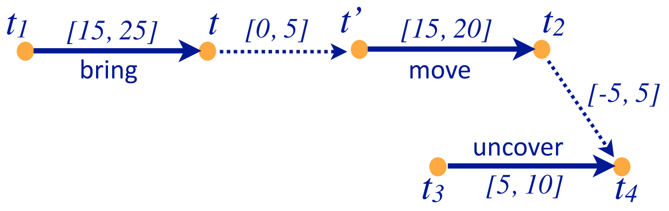


Figure 18.2. A dynamically controllable STNU.

Example 18.4. The STNU in Figure 18.1(a) is not dynamically controllable. Let us modify it by breaking bring&move in two tasks: bring from t_1 to t then move from t' to t_2, to produce the STNU shown in Figure 18.2. The total duration $[t_1, t_2]$ remains in $[30, 50]$, but the new STNU is dynamically controllable. A dynamic execution strategy for it is to trigger t_1, observe t, assign t' at any time in $[t, t+5]$ then t_3 at $t' + 10$. Regardless of the durations of the three tasks are within the bounds, the constraint $[-5, +5]$ on the end points t_2 and t_4 will be satisfied. □

How can we make sure that the STNU for a temporal a plan ϕ is dynamically controllable? It turns out that this can be done by an extension of the consistency-checking algorithm. This extension is technically involved, but fortunately it does not change the computational complexity of consistency checking.

A first step runs PC on an STNU as an ordinary STN. If the transitive update operation $(r_{ij} \leftarrow r_{ij} \cap (r_{ik} \bullet r_{kj}))$ reduces any contingent constraint, then the network is not dynamically controllable. If none is reduced, then the STNU is *pseudo-controllable*. This condition is necessary but not sufficient for dynamic controllability.

Testing dynamic controllability. Dynamic controllability can be analyzed with three basic constraints between two controllable points t_s and t and a contingent one t_e (Figure 18.3). This network is assumed to be consistent and minimal. It may or may

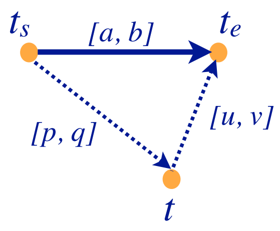


Figure 18.3. Basic constraints for dynamic controllability, where t_e and $[a, b]$ are contingent.

not be dynamically controllable: depending on the values of the parameters and the eventual observation of t_e, there may be cases in which it is possible to choose t while meeting the constraints. For that, further reductions on the controllable constraints might be needed, which conditions the dynamic controllability. These reductions would have to be propagated to other time points that may possibly be related to t_s, t_e, and t.

The position of t with respect to t_e fits into three main cases:

(i) $v < 0$: t necessarily follows t_e; the observation of t_e allows the choice of t while meeting the constraint $[u, v]$.

(ii) $u \geq 0$: t is before or simultaneous with t_e. Thus t has to be chosen before observing t_e, in an interval that meets all the constraints regardless of the value of t_e, if such an interval exists. The constraint on $[t, t_e]$ requires $t_e - v \leq t \leq t_e - u$. At the latest, t_e is such that $t_e = t_s + b$; at the earliest, $t_e = t_s + a$. Hence $t_s + b - v \leq t \leq t_s + a - u$. If this inequality can be satisfied, then the choice of t in $[b - v, u - a]$ after t_s meets all the constraints. The constraint $[p, q]$ has to be reduced to $[b - v, a - u]$. Note that $[b - v, a - u] \subseteq [p, q]$ since the network is minimal. However, $[b - v, a - u]$ can be empty, in which case the network is not dynamically controllable (see Exercise 17.17 and 17.18).

(iii) $u < 0$ and $v \geq 0$: t may either precede or follow t_e. A dynamic execution strategy should *wait* until some point, and make different choices for t depending on whether t_e has occurred. As in case (ii), t cannot be earlier than $t \geq t_s + b - v$ when t_e does not occur before. The waiting point is $t_s + b - v$. If $a < b - v$ then either $[t_s, t_e]$ occurs in $[a, b - v]$: the wait will make t follow t_e, and we are back to case (i); or $[t_s, t_e]$ occurs in $[b - v, b]$: t is before t_e which is case (ii). If $a \geq b - v$, then t_e cannot occur before the wait expires.

The preceding analysis gives the constraints to be reduced to satisfy dynamic controllability. For example $[p, q]$ is reduced to $[b - v, a - u]$ in case (ii). It exhibits a ternary *wait* relation: t should wait until either t_e or $t_s + b - v$. The trick is to consider this *wait* as a particular binary relation on the pair $[t_s, t]$: the corresponding edge in the network is labelled with a constraint denoted $\langle t_e, b - v\rangle$. Specific propagation rules for jointly handling these *wait* constraints and the normal ones in a network need to be devised.

These propagation rules are given in Figure 18.4. A row in this table is similar to the propagated contraint $(r_{ik} \bullet r_{kj})$ from i to j through k that we used in PC. The left column gives the conditions under which a propagation rule applies, and the right column states the constraint to be added to the network according to that rule. Double arrows represent contingent constraints, and angle brackets are wait constraints. The first and second rules implement, respectively, the cases (*ii*) and (*iii*). The third rule adds a lower bound constraint to a *wait*, which follows directly from the above argument. The last two rules correspond to transitive propagations of a *wait*.

Figure 18.4. Constraint propagation rules for dynamic controllability, where $a' = a - u, b' = b - v$, double arrows are contingent constraints, and $\langle t, \alpha \rangle$ are wait constraints.

Conditions	Propagated constraint
$t_s \xRightarrow{[a,b]} t_e \ , \ t \xrightarrow{[u,v]} t_e \ , \ u \geq 0$	$t_s \longrightarrow \ [b', a'] \ t$
$t_s \xRightarrow{[a,b]} t_e \ , \ t \xrightarrow{[u,v]} t_e \ , \ u < 0 \ , \ v \geq 0$	$t_s \longrightarrow \ \langle t_e, b' \rangle \ t$
$t_s \xRightarrow{[a,b]} t_e \ , \ t_s \xrightarrow{\langle t_e,u \rangle} t$	$t_s \longrightarrow \ [min\{a, u\}, \infty] \ t$
$t_s \xrightarrow{\langle t_e,b \rangle} t \ , \ t' \xrightarrow{[u,v]} t$	$t_s \longrightarrow \ \langle t_e, b' \rangle \ t'$
$t_s \xrightarrow{\langle t_e,b \rangle} t \ , \ t' \xRightarrow{[u,v]} t \ , \ t_e \neq t$	$t_s \longrightarrow \ \langle t_e, b - u \rangle \ t'$

It can be shown that a modified path consistency algorithm relying on these rules is correct: a network is dynamically controllable if and only if it is accepted by the algorithm. Furthermore, the reduced controllable constraints obtained in the final network give a dynamic execution strategy. The transposition of the *wait* constraints as a distance graph allows the incremental testing of dynamic controllability with an algorithm in $O(n^3)$.

Synthesis of dynamically controllable plans. We need to add to the requirements of Definition 17.16 a fourth condition stating that the temporal constraints in chronicle ϕ define a dynamically controllable STNU. This requirement can be met by checking dynamic controllability whenever TemPlan adds a resolver to current ϕ with a contingent constraint, and rejecting the resolver if the resulting STNU is not dynamically controllable.

This strategy can be demanding in computational resources. The complexity growth of dynamic controllability checking is polynomial, but the constant factor is high. A possible compromise is to maintain solely the pseudo-controllability of ϕ. The standard PC algorithm already tests whether a network is pseudo-controllable: no contingent constraint should be reduced during propagation. This is a necessary condition for dynamic controllability that filters out incrementally resolvers that make the STNU not pseudo-controllable. Dynamic controllability is checked before terminating with a complete solution or at a few regular stages. As for any incremental filtering strategy, the risk of excessive backtracking has to be assessed empirically.

18.2 A Dispatching Algorithm

When an actor is given a temporal plan with a dynamically controllable STNU $(\mathcal{V}, \tilde{\mathcal{V}}, \mathcal{E}, \tilde{\mathcal{E}})$, it has to trigger elements of $\mathcal{V}$ at the right moment, given the observation of elements of $\tilde{\mathcal{V}}$ and the progress of time, while keeping the rest of the network dynamically controllable. A dispatching algorithm is in charge of finding which controllable points are ready to be triggered, how to handle the wait constraints in the STNU, and how to propagate the values of observed and triggered time points. The network remains dynamically controllable as long as there is no violation of contingent constraints: observed durations of actions and expected events fit within their stated bounds. A violation of a contingent constraint can be due to a delay that exceeds the modeled upper bound, or to a failure of an action. It can lead to a failure of the plan.

In addition to end points of actions, there can be several other contingent time points in an STNU, e.g., for expected events and intermediate point such as t in the definition of leave or unstack in Example 17.11. Constraints on these contingent points are propagated (e.g., as waits) for the dynamic controllability of the network. But unless there is a wait constraint on such a contingent point, it does not interfere with dispatching and can be ignored.

Example 18.5. To follow the plan in Figure 17.4, an actor has to perform the actions leave(r1,dock1), navigate(r1,w1,w2), enter(r1,dock2) and the symmetrical actions for r2. The actor may trigger the two leave actions concurrently in any order. As soon as an exit is free, the robot moves to the corresponding way and immediately starts its navigation. When a navigation finishes, the enter action is triggered only when the other robot has left its original position. □

A temporal network is *grounded* when at least one of its temporal variables receives an absolute value with respect to current time. Before starting the execution, the STNU may or may not be grounded, but as soon as the execution of a plan starts, the network is necessarily grounded. In a grounded network, every time point t is bounded within an absolute interval $[l_t, u_t]$ with respect to the current time, which we will denote by *now*. As time goes by, some time points in the network have occurred (i.e., triggered by the actor for controllable points or observed for contingent ones), and others remain in the future. Dispatching is concerned with the latter and more precisely with enabled time points.

Definition 18.6. A controllable time point $t \in [l_t, u_t]$ that remains in the future is *alive* if the current time $now \in [l_t, u_t]$. Furthermore, t is *enabled* if (*i*) t is alive, (*ii*) for every precedence constraints $t' < t$, t' has occurred, and (*iii*) for every *wait* constraint $\langle t_e, \alpha \rangle$, either t_e has occurred or α has expired. □

Recall that in a wait constraint $\langle t_e, \alpha \rangle$, α is defined with respect to a controllable time point t_s. Thus α has expired when t_s has occurred and $t_s + \alpha \leq now$.

The Dispatch algorithm controls when to start each action. In Line 1, it sets *now* and triggers when to start the plan, which grounds the network if it is not already grounded. It then triggers enabled points whose upper bound is *now*; these cannot wait longer. Dispatch has the flexibility to trigger any other enabled point. The arbitrary choice in Line 3 can be made with respect to domain-specific considerations. The values of the triggered points are then propagated in the network. Because the STNU is dynamically controllable, this propagation is guaranteed to succeed and keep the network dynamically controllable as long as contingent constraints are not violated. Monitoring that no such violation occurs can be added in Line 2 (see Exercise 18.1)

```
Dispatch($\mathcal{V}, \tilde{\mathcal{V}}, \mathcal{E}, \tilde{\mathcal{E}}$)
1  initialize the network
   while there are elements in $\mathcal{V}$ that have not occurred do
       update $now$
2      update contingent points in $\tilde{\mathcal{V}}$ that have been observed
       $enabled \leftarrow$ set of enabled time points
       foreach $t \in enabled$ such that $now = u_t$ do
           trigger $t$
3      arbitrarily choose other points in $enabled$ and trigger them
       propagate in the network the values of triggered points
```

Algorithm 18.1. Dispatch, a dispatching algorithm.

The propagation step is the most costly one in Dispatch. Its complexity is in $O(n^3)$, where n is the number of remaining future points in the network. Ideally, this propagation should be fast enough to allow iterations and updates of *now* that are consistent with the temporal granularity of the plan. As discussed next about action refinement, this step is faster when actions are not at too fine a granularity.

Example 18.7. Let us extend Example 18.5 by requiring robot r1 to bring a container c1 from dock d2 to some destination. Figure 18.5 shows part of the plan. For simplicity, the values of the constraints and parameters are omitted; the end point of an action starting at t_i is implicitly named t'_i. Note that some of the object variables are instantiated, but some are not (e.g., c'); temporal variables in ϕ are not instantiated.

The initial step in Dispatch triggers t_1. When t'_1 is observed, t_2 is enabled and triggered, which make t_3 and t_4 enabled. The algorithm triggers t_3 far enough in advance to free dock d2 so that r1 can get in (at t_5). Similarly for the subtask of uncovering container c, which is triggered at t_4. When t'_2 and t'_3 are observed, t_5 becomes enabled and triggered. When t'_5 and t'_6 are observed, this enables t_7. The rest of the plan follows linearly. □

Let us now consider how to integrate dispatching into an actor's deliberations.

18.3 Acting without Temporal Refinement

We consider here successively two acting modalities when given a temporal plan: without any refinement, then with atemporal refinement.

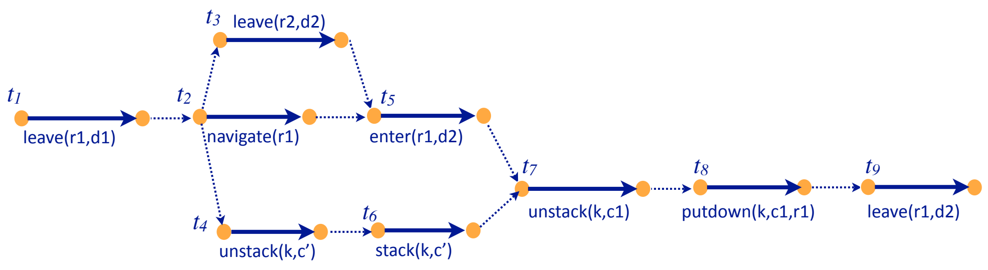


Figure 18.5. Dispatching of part of a temporal plan.

18.3.1 Acting with a Detailed Plan without Refinement

Suppose we are given a temporal plan ϕ with a dynamically controllable STNU whose actions are primitives to be executed on the platform without refinement. The actor needs a simple execution engine without refinement, which has only actions and events in its input agenda, with deadline handling mechanisms.

The Dispatch algorithm is easily integrated into this engine. Triggering an action a means putting a in the input agenda. The upper bound on the duration of a is taken as a deadline for terminating a. Action failures have to be addressed as plan repairs. For a deadline failure, the repair can take two forms:

- stopping the delayed action and seeking alternate ways to achieve the plan from the current state, as for other types of failure, or
- finishing the action despite the delay, and repairing the remaining part of the plan by taking in account the delay.

The latter option is preferable when the violated contingent constraint can be resolved at the STNU propagation level. For example, if navigate(r1) in Figure 18.5 takes slightly longer than the maximum duration specified, the entire plan will still be feasible with a delay, which is possibly acceptable. However, if this navigation is taking longer than expected because robot r1 broke down, a better option is to seek another robot to perform the task. Such domain-specific considerations must be integrated into the actor's monitoring function.

Plan repair in case of a failure has to be performed with respect to the current state, remaining predicted events, and tasks whose achievement is still in the future. The repair can be local or global. In the latter case, full replanning is performed. A local repair can benefit from TemPlan's plan-space planning approach, by removing the failed action from the remaining chronicle ϕ together with all the assertions coming from that action schema. This removal introduces flaws in ϕ with respect to which TemPlan iterates. This can lead to other flaws (including for the task that led to the failure). It may or may not succeed in finding a repair and may require replanning. Monitoring should help assess the failure and decide whether to try a local repair.

18.3.2 Acting with Atemporal Refinement Methods

Here we assume that actions in the given temporal plan ϕ need to be further refined by the actor. For example, actions leave, enter, stack and unstack in Example 17.11 are primitives for TemPlan, but they need to be refined into executable commands with appropriate refinement methods for the acting context. If instead we had planned with these lower level primitives, this would have increased the planning complexity, required finer action models and made the plan fragile.

Hence, acting refinement is motivated by complexity, environment dynamics, and modeling issues. It goes one level down in the representation hierarchy. In many cases, an actor may refine the actions in a temporal plan with *atemporal* methods, meaning that the duration of an action a in the given plan is not broken down further with this acting refinement.

Example 18.8. Here are two atemporal methods to refine actions leave and unstack of Example 17.11 into commands:

```
m-leave$(r, d, w, e)$
    task: leave$(r, d, w)$
     pre: loc$(r) = d$, adjacent$(d, w)$, exit$(e, d, w)$
    body: until empty$(e)$ wait$(1)$
          goto$(r, e)$

m-unstack$(k, c, p)$
    task: unstack$(k, c, p)$
     pre: pos$(c) = p$, top$(p) = c$, grip$(k) =$ empty
          attached$(k, d)$, attached$(p, d)$
    body: locate-grasp-position$(k, c, p)$
          move-to-grasp-position$(k, c, p)$
          grasp$(k, c, p)$
          until firm-grasp$(k, c, p)$ ensure-grasp$(k, c, p)$
          lift-vertically$(k, c, p)$
          move-to-neutral-position$(k, c, p)$
```

The method m-leave waits until the exit e from dock d toward waypoint w is empty, then it moves the robot to that exit. The method m-unstack locates the grasping position for container c on top of a pile p, moves the crane to that position, grasps it, ensures the grasp (e.g., closes latches) to guarantee a firm grasp, raises the container slowly above the pile, then moves away to the neutral position of that crane.These methods have not temporal variables. They handle time implicitly through the procedures in their body.

It is interesting to compare these methods with the descriptive models of the same primitives in Example 17.11. Effects are not stated; they will be observed from the execution of commands. The operational models given in these methods detail with conditionals and loops how to perform the action. □

Atemporal refinement methods do not break down the temporal qualifications used in planning into finer temporal requirements. As illustrated in the preceding example, the temporal qualification $[t_s, t_e]$ of an action a in ϕ is not detailed into smaller durations for the commands in which a is refined.

An important motivation for refining a temporal plan with atemporal methods, instead of temporal ones, is the uncertainty in the duration of actions. It makes sense to reason about contingent constraints at an abstract planning level. But at the lower fine grained level, one may take into account a global constraint without refining it into bounds that can be even more uncertain, more difficult to model and control in a meaningful way. For example, it may be useful to account for the time needed to open a door, which can be assessed from statistics. However, breaking this duration into how long it takes to reach for the handle and how long to turn the handle introduces noisy data in operational models.

Chapter 14 explains how to refine tasks with atemporal methods. To follow a temporal plan ϕ by refining its actions with atemporal methods, the actor can use RAE augmented with the Dispatch algorithm. The latter is easily integrated with RAE. Triggering the starting point of an action a means putting a as a task in the input agenda of RAE, that is, starting a new stack for progressing on the refinement of a. The upper bound on the duration of a is taken as a deadline for terminating this stack. Progress and eventually Retry will pursue refinements in this stack until the action succeeds or fails, or until the deadline is reached, which is another failure condition. The distance to the deadline can be used as a heuristics for prioritizing the most urgent tasks.

18.4 A Temporal Refinement Acting Engine

In the previous section we discussed two simple acting modalities: without refinement and with atemporal refinement. Here we address the more complex modality of temporal refinement, with or without a plan. An actor may not have a plan for the task at hand. This may happen because a plan is implicit in the available methods of task, or because the descriptive models of actions are unreliable, or the environment is too dynamic and acting with retrials is not critical. Without a temporal plan, it is still meaningful to reason about time at the acting level, in particular when acting has to be synchronized with future predicted event.

Temporal refinement methods can be used for acting. For that, we restrict the general representation of Section 17.1.3 to methods that do not specify unsupported assertions, i.e., goals requiring generative planning.

The approach is similar to that of Chapter 14 where the acting engine reacted to tasks using refinement methods. Here we modify RAE as TRAE to obtain a temporal refinement acting engine. The refinement methods used by this engine differ from those of RAE in a few important ways:

- The methods' bodies are not procedures with tasks, actions, assignments, and control steps, but instead are sequences of temporally qualified tasks, assertions, and constraints.
- They do not have specific precondition fields. Instead, their preconditions result from temporal assertions (e.g., as in the methods of Example 17.12).

- They do not deal with probabilistic actions, but deterministic ones with uncertain durations and contingent expected events.
- They do not transform the current state sequentially, step by step, but they progress on a time line by triggering and observing concurrent changes.

RAE evaluates a conditional expression with respect to the current observed state. TRAE requires an acting context integrating the current state to timelines with future predicted exogenous variables and past values of state variables. For example, a method may allow a robot to visit a location l1 only if it has gone through a location l2 in the recent past. Instead of expressing this and similar constraints as temporal assertions extending over the past history, we may express them with additional state variables.[2]

With this modeling caveat, let us assume that the acting context does not extend in the past beyond when each state variable acquired its current value. Our interest in the past is satisfied by keeping track for each state variable x that the value $x = v$ *since* holds from t to *now*: In other words, the following assertion holds:

$$[t, now]x = v, \text{ where } now \text{ it the value of current time.}$$

Note that such an extended state allows the handling of assertions such as $[t_1, t_2]$loc(r1)=l2, $(t_2 - t_1) > \delta$.

The acting context for TRAE corresponds to a chronicle ϕ_ξ giving the current present and the predicted exogenous future. ϕ_ξ is similar to the initial chronicle ϕ_0 of TemPlan with an essential difference. ϕ_0 contains tasks to refine as well as assertions to support, which may express goals to reach, in addition to the effects of the refined tasks. Unsupported assertions are allowed by the hybrid TemPlan since it is a refinement as well as a generative planner. But TRAE is only a refinement acting engine. It works similarly to RAE: it is given only tasks to perform, but no goals to reach, and it uses methods that do not introduce additional goals to reach.

TRAE gets as input the tasks to perform. For each task τ, it finds $\mathcal{M}(\tau)$, the set of methods whose task is τ. It chooses a method m in $\mathcal{M}(\tau)$ that is applicable to the current state ξ. It refines τ according to the subtasks specified in m. This is performed similarly to resolving for an unrefined task in TemPlan, with an update $\phi_\xi \leftarrow$ Transform(ϕ_ξ, m): ϕ_ξ is augmented with the subtasks, assertions and constraints in m (see Section 17.2.2). The requirements for a method to be applicable are however different here.

Definition 18.9. An instance in a method $m \in \mathcal{M}(\tau)$ is applicable to the current acting state ξ if and only if:

- every assertion in m is causally supported by and consistent with ξ, and
- the constraints in m are consistent with ξ and define with the temporal constraints in ϕ_ξ a dynamically controllable STNU. □

The first condition requires that m does not introduce goals to reach. For example, an assertions in m such as $[t, t']\, x = v$ or $[t, t']\, x : (v, v')$, where $t < t_s \leq t'$ and t_s is

[2] This is akin to how we may model a domain as Markovian (see Example 8.1).

the starting point of m, would need to be supported before any action issued from the refinement of m can begin. Assertions in m about the future have to be supported by predictions in ϕ_ξ. The acting engine is not inserting (in a generative way) additional actions to satisfy the requirements of a method. Assertions that require to be supported by the effects of *additional* actions are not allowed in the methods of TRAE.[3]

The second condition guarantees that the application of this instance of m to ξ gives a consistent chronicle whose temporal constraints are dynamically controllable. The latter point is only about the explicit constraints in Transform(ϕ_ξ, m). The recursive refinements of the subtasks in m may lead later to an uncontrollable network, but TRAE cannot detect it in advance.

Example 18.10. Consider the domain specified in Example 17.11. Assume that TRAE is given a set of methods to handle the tasks bring, move, and uncover of Example 17.12, in addition to methods for leave, enter, navigate, unload, load, stack, and unstack, as illustrated in the previous section.

The task is to bring a container c1, which is *now* in pile p1 in dock d1, to a pile p2 in d2. There is an empty robot r1 in d3. An instance of the method m-bring is applicable to this task with $c =$ c1, $p =$ p2, $p' =$ p1, $d =$ d2, $d' =$ d1, $r =$ r1; t_s can be instantiated to *now*: TRAE concurrently triggers the tasks move(r1,d1) and uncover(c1). The constraints $t_2 \leq t_1$ and $t_3 \leq t_1$ require the other tasks to wait until both move and uncover finish.

The method m-move1 is applicable to move(r1,d1). TRAE triggers action leave. When leave finishes, navigate is triggered, followed by enter.

Concurrently with this process, TRAE addresses the uncover task: method m-uncover is applicable; it leads to triggering a succession of unstack and stack actions until d1 is at the top of pile p1. The termination of the last actions issued from the refinement of move(r1,d1) and uncover(c1) set the time points t_2 and t_3 of m-bring, allowing the method to pursue the remaining subtasks load, move and unload. □

To sum up, TRAE uses on temporal refinement methods without unsupported assertions. It takes as input tasks to perform. It maintains an acting context corresponding to a chronicle ϕ_ξ with the current state and the predicted exogenous events.

To progress in the refinement of a method m, TRAE has to handle concurrency, as specified in m, and to monitor observed expected changes for contingent time points. This is done with the Dispatch algorithm for the chronicle ϕ_ξ, which triggers controllable time points in ϕ_ξ that are *enabled* with respect to *now*. Contingent points in ϕ_ξ (termination of actions, expected events) are monitored with a deadline corresponding to the wait constraints introduced while testing the applicability of m.

The refinement of m by TRAE may fail when either (*i*) a triggered action fails, (*ii*) a wait constraint on a contingent point fails, or (*iii*) a refinement of a subtask τ' in m fails because there is no applicable method for τ' or all method instances have been tried and failed. Cases (*i*) and (*iii*) are as in RAE. Case (*ii*) is specific to dynamic controllability when contingent constraints or deadlines are violated. TRAE can implement a Retry procedure for trying alternative methods if the chosen one

[3] A less restrictive Definition 18.9 would allow in m assertions and constraints to be supported by the subtasks and actions of m, but checking if m is applicable would be difficult.

fails: a Retry is possible as long as upper bounds on a task and its refinements have not been violated.

The limitations of TRAE are due to its lack of lookahead. As underlined earlier TRAE does not handle reachability goals, but only task refinements. Moreover, the network corresponding to the entire refinement tree of a task τ and its subtasks is not guaranteed to be dynamically controllable. Additional points and constraints are discovered progressively by TRAE as tasks and actions are achieved, and effects and events are observed. This may lead to situations where no method for a subtask gives a controllable STNU. The designer may try to specify (with the help of the techniques presented in Section 18.1) temporal methods that often refine into dynamically controllable networks. But it is easier and preferable to endow TRAE with a lookahead mechanism, as presented next.

18.5 An MCTS Temporal Planner

We study here how to transpose the lookahead mechanism of RAE to TRAE. The former relied on UPOM that we transpose here to UTEMP. Recall that UPOM performs Monte Carlo rollouts with value updates to optimize a value function and guide the choices of RAE with good methods. UPOM relies on two essential means: a simulation of method bodies, and a sampling function that returns a possible result of probabilistic actions.

These two means are feasible for TRAE. Simulation of temporal refinement methods is sightly more complex because of concurrency and the dispatching mechanism. The sampling function is about sampling the outcome of actions as success or failure, and contingent time points within allowed intervals.

Let us informally present the MCTS planner UTEMP by comparison to UPOM.

Common features of UTEMP and UPOM. These are the following:

- they work with a set of states S which is an abstraction of the detailed action states Ξ;
- they update a value function $Q(s, m)$ initialized heuristically;
- they use a procedure Guide which improves Q through a number of n_{ro} progressively deeper rollouts;
- they are anytime planners;
- they progress in each rollout by managing a stack initialized to the current acting stack$_a$;
- they rely on a UCT-like procedure to update the value function Q along a rollout for the stack at hand.

The specifics of UTEMP with respect to UPOM are about (*i*) the search space and the stack data structure, (*ii*) the temporal dispatching in the progress of a rollout, (*iii*) the utility and value functions, and *(iv)* the Sample function.

Search space of UTEMP. A state s in UTEMP is temporally extended. As in TRAE, it corresponds to a chronicle ϕ_s giving the current present and predicted future. To simulate in a rollout a method m, ϕ_s is augmented with the subtasks, assertions and constraints in m, with the update $\phi_s \leftarrow \text{Transform}(\phi_s, m)$ explained earlier.

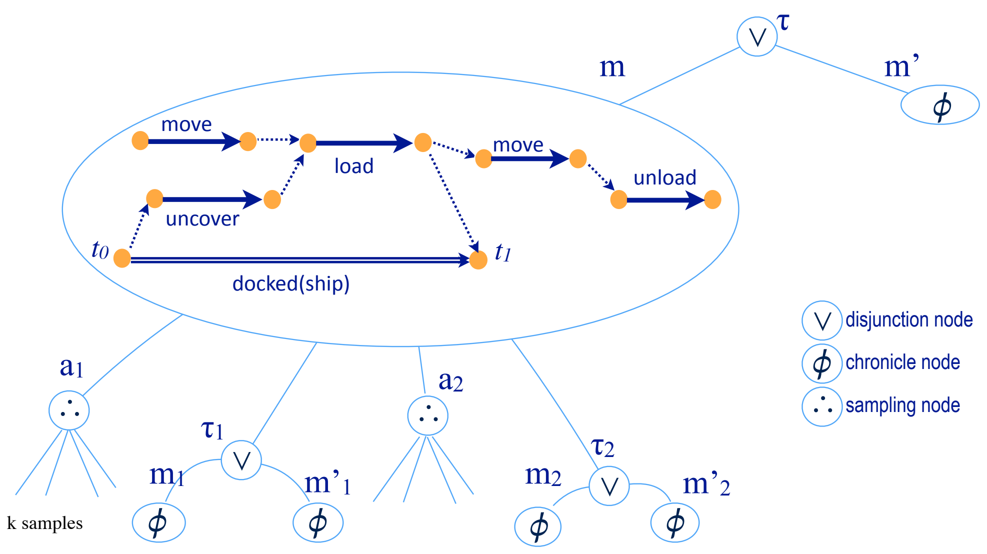


Figure 18.6. Part of the search space for UTEMP, *disjunction* nodes for possible methods tasks, *chronicle* nodes for method instances with their temporally qualified tasks assertions and constraints, and *sampling* nodes for possible occurrences of contingent points. Solid arrows are tasks and actions, double are contingent activities, dotted are precedences and wait constraints; all are labelled with temporal contraints is in Figure 18.2.

While UPOM handles a sequence of steps in a method body to simulate that method in a rollout, UTEMP has to handle a sequence of chronicle updates. In UPOM, the stack is a list of tuples $(\tau, m, i, \mathit{tried})$, where i is an index for the current step in m. In UTEMP a stack tuple is of the form $(\tau, m, \phi_s, \mathit{tried})$ where ϕ_s is the current chronicle corresponding to m for the current state and refinement step. The search space of UTEMP, illustrated next, has disjunction, chronicle and sampling nodes (Figures 18.6 for UTEMP *vs* 15.1 for UPOM).

Example 18.11. Consider the task bring of Example 17.12 addressed with the method m-bring in a state corresponding to the chronicle ϕ_0 of Example 17.14. ϕ_0 specifies that the container to bring is on board of a ship to be docked in a given temporal interval. The chronicle Transform(ϕ_0, m-bring) is simulated using Dispatch, taking the starting time t_s as the beginning of the simulation, and advancing the current time *now* by instantaneous steps to the next enabled points. Action move is triggered immediately, the task uncover is refined after t_0, the load action after uncover and move, etc. At some point of a simulation, a rollout may fail through random sampling for the same reasons as the possible failures of TRAE: a failed action or task refinement, or a violated constraint on a contingent point. □

Temporal dispatching in the progress of a rollout. UTEMP progresses in a rollout simulation as TRAE in real task refinement and acting. It uses Dispatch and triggers all enabled points at the earliest. There is however an important difference between the two with respect to expected events grounded in real time (e.g., t_0 and t_1 in Figure 18.6). We need to decouple simulated time from real time. This can be done by estimating in advance a worst case duration of a task τ, from which a deadline for finishing UTEMP on a lookahead for τ is computed. Since UTEMP is an anytime planner, as soon as the estimated deadline is reached, the deepening loop in Guide is stoped and TRAE proceeds with the best method found so far.

Optimization criteria. The utility and value functions of UPOM can also be used for UTEMP. We defined U and its estimate Q using either v_e, the efficiency of an action, or v_s, its success rate. Equation 15.1 applies to UTEMP with a minor change (there are no assignments in temporal methods). However, these criteria are not quite adequate for temporal models, for which one is usually interested in minimizing a plan's makespan or maximizing the slack time on the plan's critical path, i.e., how much the actions can be delayed without violating the plan constraints. An extension of v_e taking into account the duration of an action as well as its cost is straightforward. However, this does not provide the makespan nor slack time of a rollout. These criteria can only be roughly approximated with cumulative updates on partial rollouts. They can be computed precisely on complete ones using the well-known techniques of project network analysis.[4] Implementing such an idea for minimizing the makespan or the slack time requires revising in UPOM pseudocode the definition of U_{Success} and the termination step (when stack = $\langle\rangle$) for UTEMP.

Sampling function. UTEMP samples on actions and contingent events. When an action a is triggered, Sample(s, a) returns randomly a finish time and either failure or s' for the successful completion of the assertions in a. As usual, we assume that calls to Sample are randomly distributed according to the probability distribution characterizing a. Here too one may draw on project network techniques for defining meaningful temporal distributions for an expected duration within the earliest and latest finish times. The sampling of contingent event can be done on the initial chronicle, before a rollout starts. It returns random occurence times within predicted bounds of the events and their corresponding assertions. It is easier to assume that predictions are deterministic but, if needed, one also sample for a possible nonoccurrence of a predicted event, which may lead to failure.

It can be interesting to complement the MCTS strategy with non-random rollouts that are meaningful for temporal domains. For example, temporally "pessimistic" rollouts with worst case durations and event occurrences may allow seeking robust methods, instead of near-optimal ones in average.

As underlined earlier, to our knowledge TRAE and UTEMP have not been implemented and tested for scalability and performance issues. TRAE should be effective, since it is a reactive engine. Its costly steps for testing the applicability of methods and running Guide, involve only low order polynomial computations. Similarly for each rollout of UTEMP. In addition, controllabily computations along a rollout may

[4]These are often referred to as *Program Evaluation and Review Techniques* (PERT).

be postponed, as explained in page 418. However, it is unclear how effective UTEMP can be for a domain requiring a very large amount of rollouts. Learning for UTEMP (discussed in the following chapter) should be effective. The interested reader is encouraged to consider the material in this section as an invitation to further deepen the proposed approach into detail code and analysis.

18.6 Integrating Planning and Acting

Previous sections present three main integration modalities of planning and acting:

- Planning with TemPlan with acting without refinement;
- Planning with TemPlan with acting atemporal refinement methods; and
- Acting with TRAE with UTEMP lookaheads.

The first one is for well modeled critical domains with low variability. The last one is more reactive. It can better cope with variability. Moreover, TemPlan is a satisfycing approach with correctness and controllability properties. UTEMP is an optimizing approximation approach, with no guarantee, but its any time property is important in temporal domains.

These integration modalities are not exclusive. If time allows, one may plan at a sufficient granularity level for the task at hand using TemPlan, then act with TRAE using refinement methods and the dynamically controllable STNU found at the planning stage. Here, TRAE does not need to test the applicability of its methods with the restrictive Definition 18.9. This testing is done at planning time by adding, when and where needed, actions in the plan to support every assertions in the predicted future. TRAE has to monitor that the current acting state is the one expected at planning time. It has also to synchronize the subtasks and actions in the plan with Dispatch according to the constraints in the dynamically controllable STNU. In case of departure from the plan because of failure or violation of a contingent constraints, TRAE may resort UTEMP to guide its reactive choices.

The main difference with respect to following a plan with atemporal refinements is that here one is able to control the level at which refinement planning is pursued in a context-dependent way. The idea is to allow TemPlan to decide not refine a subtask. This can be done if TempPlan can evaluate the likely effects and temporal bounds of that subtask and assess that they are sufficient to stop planning and start acting on a partial plan that contains unrefined tasks. These can be refined at acting time, by planning concurrently while acting on some other predecessor subtasks, or even when an unrefined task is dispatched. In this modality, the interactions between planning and acting are however more complex than the on-demand and anytime guiding mechanisms of TRAE/UTEMP.

18.7 Discussion and Bibliographic Notes

Temporal controllability. The controllability issue and STNUs was introduced in [1139]. Different levels of strong, weak and dynamic controllability were analyzed

[1138]. Informally, an STNU is strongly controllable when the actor can freely chose the assignments of controllable points in their bounds, and regardless of the choices of the world for the contingent ones the execution strategy (in p.415) exit with success. An STNU is weakly controllable when there exist values of contingent points for which there are assignment of controllable ones that allow a successful exit. The dynamic controllability, which allows the actor to adapt to its observations and wait when needed, is more interesting in practice. Algorithms for the strong and weak controllability cases appears in [239, 238]. State space planning with strong controllability is studied in [240]. A polynomial algorithm for dynamic controllability was proposed by [810] and improved in [812]. Incremental dynamic controllability has been studied in [1055], and [854, 855] for the algorithm of cubic complexity.

Project networks. These are network of activities and events with constraints and uncertainty in durations and occurrences. They are like chronicles but without a generic representation for assertion, tasks and actions. A complete rollout of UTEMP (without unrefined tasks) is in similar to a project network. Several techniques and numerous software libraries have been developed for analyzing the properties of a project network, finding its critical path, makespan or slack time, see e.g., [599].

Acting with Temporal Models. Several of the acting representations and systems discussed in Section 14.4, based on procedures, rules, automata, Petri-nets or CSPs, integrates directly or have been extended with temporal primitives and techniques for handling explicit time. The PRS system of [540] or the RPL language of [774] offer some mechanisms for handling real-time "watchdogs" and delay primitives. More elaborate synchronization constructs are available in TCA [1025] and TDL [1026].

A few of the temporal planners discussed in paragraph 17.4 have been integrated to and acting system. This is in particular the case for timeline oriented planners along an approach akin to that of Section 18.3.2. For example, Cypress of [1173] is the combination of SIPE for planning and PRS for acting. DS1/RAX of [823] implements a procedure-based acting technique combined with the PS planner. Casper of [616] is a temporal constraint-based executor for the ASPEN planner. IxTeT-Exec of [694] integrates IxTeT and PRS with plan repair and action refinement mechanisms. T-REX of [929] follows a distributed approach over a set of "reactors" sharing timelines. It has been used mostly with the EUROPA planner. The *dispatchability property* studied in [822] and [811] requires simplifying the STNs resulting from the above planners in order to rely on local propagation at acting time. This technique provides some improvements in the dispatching algorithm but does not handle dynamic controllability.

The Reactive Model-based Programming Language (RMPL) of [536] follows an approach more akin to that of Section 18.4. RMPL programs are transformed into the Temporal Plan Networks (TPN) representation of [1174]. TPN extends STN with symbolic constraints and decision nodes. Planning with a TPN is finding a path in the explicit network that meets the constraints. [255] introduce choices in the acting component of RMPL. TPNs with error recovery, temporal flexibility, and conditional context dependent execution are considered in [322]. There, tasks have random variable durations with probability distributions. A particle-sampling dynamic execution algorithm finds an execution guaranteed to succeed with a given probability. Probabilistic TPNs with weak and strong consistency are proposed in [980]. TPNUs of [702] add the notion of uncertainty for contingent decisions taken by the environment and other agents. The acting system adapts the execution to observations and predictions based on the plan. It has been illustrated with a service robot which observes and assists a human.

18.8 Exercises

18.1. Revise the pseudocode of algorithm Dispatch by adding a monitoring step to check that contingent constraints are not violated.

18.2. Detail the pseudo-code of TRAE specified in Section 18.4 as an extension of RAE (see Chapter 14).

18.3. Detail the pseudo-code of UTEMP specified in Section 18.4 as an extension of UPOM (see Chapter 15).

19 Learning for Temporal Acting and Planning

Temporal models are quite rich. This an advantage for handling domains with concurrency and temporal constraints. But this is also a bottleneck for the development of the models, to be eased with machine learning techniques.

In this chapter, we first briefly address the problem of learning heuristics for temporal planning (Section 19.1). We then consider the issue of learning durative action schema and temporal methods (Section 19.2). The chapter outlines the proposed approaches, based on techniques seen earlier in the book, without getting into detailed description of the corresponding procedures.

19.1 Learning Heuristics for Temporal Planning

This section sketches how to learn heuristics for guiding TRAE and UTEMP, following the approach used for RAE/UPOM (see Chapter 16).

Recall that TRAE relies on Guide to choose its methods. If there is no time to call UTEMP, Guide returns a fallback method $\tilde{m} = \text{argmax}_m Q_0(s, m)$, where Q_0 is a heuristic estimate of the method-value function. Otherwise UTEMP performs a series of progressively deeper rollouts. At the end of each rollout (when $d = 0$), UTEMP uses $Q_0(s, m)$ as an estimate of the utility of the remaining refinements to compute the method-value function $Q_{\mathsf{stack}}(s, m)$ for the methods applicable to the task at hand and its subtasks. The function Q_0 approximates the method-value function Q for a state, a method and its corresponding task. It ignores the details of pending activities in a stack, if any. Q_0 is of direct use for acting reactively in Guide, as well as for planning with UTEMP. Learning a domain-dependent heuristic function Q_0 can significantly improve the performance of the acting engine as well as those of the planner.

Since we defined TRAE and UTEMP as adaptations of RAE and UPOM, we can adapt the learning approach for RAE/UPOM to learn Q_0 for TRAE/UTEMP. The following adaptations are needed:

- *Training set generation*: we randomly generate a set of planning problems (τ, s) with a temporally extended state s as an initial chronicle. It can be more difficult than in the atemporal case to come up with relevant problems solely from random sampling. UTEMP is run for each training problem and applicable method. The found near-optimal value of $Q_{\mathsf{stack}}(s, m)$ is kept as a learning target. Initial knowledge about interesting scenarios to learn from, or a rich history of previously encountered cases, are needed, to be completed with continual learning.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- *Data encoding*: the adaptions here are about encoding temporal qualifications and temporal constraints as input to a neural net. The latter does not check the consistency of the constraints nor solves them; it simply learn a mapping from (τ, s, m) to Q_0. In principle, the encoding of symbolic variables of Section 16.1.2 can be used here, possibly with add-hoc functions that have been developed for interval constraints. It can also be worthwhile to investigate the use of temporal graph neural networks.
- *Neural Net Training*: this stage is the same as discussed for RAE/UPOM if a feedforward architecture is used. We briefly refer in the discussion section to features specific to graph neural nets, in particular the pairwise message-passing mechanism.

The learned network $[Q_\theta]$ resulting from the previous steps can be maintained and improved in a continual learning framework. The reinforcement learning CORL procedure applies to TRAE/UTEMP.

The approach briefly sketched here illustrates a possible method for learning heuristics for temporal refinement acting and Monte Carlo planning. To our knowledge, it remains to be implemented and tested in order to assess how effective it can be, and in what categories of domains.

19.2 Learning Temporal Models

In this section, we briefly discuss two problems: learning durative action schema by extending the approach presented in Section 4.2, and learning temporal methods with the HTN learning techniques of Chapter 7.

19.2.1 Learning Durative Actions

This section deals with the problem of synthesizing durative action schema in a restricted case allowing to rely on an extension of the methods of Chapter 4. The learner gets as input time-stamped traces without overlapping or concurrent actions. We assume that the preconditions of an action apply at its starting time; its effects hold at the end point. Its duration is a scalar in some observed interval (not a function, to be learned, of the action parameters). We therefore rely on a more limited representation than the one based on timelines and chronicles, with state variables that may change over time (as in Section 17.1).

Moreover, we focus on offline learning. Online learning requires a temporal extension of informative state-action pairs (see Algorithm 4.12), and the use of temporal planning to learn action schemas. These issues remain an open problem.

Let t_s and t_e be the start and end time points of an action. Preconditions hold at t_s and remain true until t_e; and effects hold at t_e. We assume an action duration to be constant. With these assumptions, action schema are a simple extension of the classical case (see Sections 2.3.2 and 4.2).

We learn primitive action models from time-stamped traces, ignoring time in a first stage, seeking only preconditions and effects. Since we assume that the training traces do not have concurrent or overlapping actions, the offline methods of Section 4.2.1 are applicable. In a second stage, time bounds are estimated from training traces.

The input to the offline learner is a set of time-stamped transitions $(t, s, a(\boldsymbol{c}), s', t')$, where t and t' extend the $(s, a(\boldsymbol{c}), s')$ transition of Section 4.2.1 with the start and end time of this instance of a. Algorithm 19.1, Offline-Learning-Actions-with-Time, is an extension of Algorithm 4.6. It computes the duration of an action a as the maximum of the durations that have been observed in the set of time-stamped transitions.

```
Offline-Learning-Actions-with-Time($T$)
    for $a$ action name that appears in $T$ do
        $\text{pre}(a(z)) \leftarrow \text{U}$
        $\text{eff}(a(z)) \leftarrow \varnothing$
        $\Delta_t(a) \leftarrow 0$
    while $T \neq \varnothing$ do
        choose $(t, s, a(\boldsymbol{c}), s', t') \in T$
        $\text{pre}(a(z)) \leftarrow \text{pre}(a(z)) \cap s(z)$
        $\text{eff}(a(z)) \leftarrow \text{eff}(a(z)) \cup s'(z) \setminus s(z)$
        $\Delta_t(a) \leftarrow \max(\Delta_t(a), t' - t)$
        $T \leftarrow T \cap (t, s, a(\boldsymbol{c}), s', t')$
```

Algorithm 19.1. Offline-Action-Learning-with-Time, a simple algorithm

Algorithm 19.1 can deal with preconditions that hold only at the starting time t_s and may change before the ending time t_e. It should be extended to deal with overlapping concurrent actions in order to detect mutually exclusive actions, or actions that can be applied only before, during or after the application of other actions. In an offline approach, we can detect only the positive cases, e.g., that an action can be executed overlapping with another one, or before, or after another one, and assume that if we do not have actions that overlap, they cannot be executed in parallel. Moreover, we should take into account the possible influences of preconditions and effects of actions that overlap (see Exercise 19.3).

19.2.2 Learning Temporal Methods

This section sketches possible approaches for learning temporal refinement methods. We use a simpler representation than the general one in Section 17.1.3. Specifically, we consider temporal HTN methods in one of the following two forms:

- *Basic temporal HTN methods*: these are just like totally ordered HTN methods, but with durative actions.
- *Extended temporal HTN methods*: these methods are specified with Simple Temporal Networks (STNs) constraining possible refinements and allowing for concurrency.

Learning basic temporal HTN methods. We use the techniques of Chapter 7. We assume we are given:

- A set A of durative actions represented with their descriptive models, as learned with the approach of previous section.
- A set $\mathcal{T}$ of annotated tasks $\tau = (\text{task}(\tau), \text{pre}(\tau), \text{eff}(\tau))$, as in Chapter 7.
- A temporal planner such as TemPlan that can generate temporal plans using the actions in A to reach goals.

The learner has to synthesize a set $\mathcal{M}$ of temporal methods for the tasks in $\mathcal{T}$. The outline of the approach is the following:

- Randomly generate a set of planning problems $(s_0, A, \text{eff}(\tau))$, for each $\tau \in \mathcal{T}$ and for a collection of states $s_0 \in \text{pre}(\tau)$.
- Run the planner on each problem $(s_0, A, \text{eff}(\tau))$. If a plan π achieving $\text{eff}(\tau)$ from s_0 is found, record the corresponding pair (s_0, π) in the training set *Pairs*.
- Run a version of Methods-from-Plans on the set *Pairs* to obtain $\mathcal{M}$.
- Check if the learned methods $\mathcal{M}$ allow planning any newly generated random problem (s_0, τ), for $\tau \in \mathcal{T}$ and $s_0 \in \text{pre}(\tau)$; extend $\mathcal{M}$ if needed.

Here, an incremental version of Methods-from-Plans integrating the steps of Methods-from-Examples (instead of calling it on a collection E of examples for all the generated plans) would be desirable. This can be particularly beneficial if some knowledge about the task hierarchy is given or can be derived from the desired effects $\text{eff}(\tau)$ for $\tau \in \mathcal{T}$. Learning would proceed bottom-up in this hierarchy. Learned methods of subtasks would be used for planning higher level tasks (recall that a hybrid planner such as TemPlan performs task refinement and goal reachability searches), as well as for learning their methods. Since basic temporal HTNs assume totally ordered decompositions, the temporal part with durative action plays no role in learning and can be processed in a straightforward manner on learned methods.

Learning extended temporal HTN methods. Here, we want to synthesize methods similar to those of partially ordered HTNs, but instead of a partial order of the steps in a method, we specify a Simple Temporal Network (STN) on the starting and ending points of these steps, as illustrated next.

Example 19.1. Consider the method m1-put-on-robot of Example 5.10. It refines a task put-on-robot into three steps (t1, navigate), (t2, unstack) and (t3, load). The latter starts only when the first two finish: t1 < t3, t2 < t3. But for an efficient handling of the task, we do not want to start unstacking too late (it keeps the robot waiting), or too early with respect to the possibly longer navigation step. This can be expressed with an STN on the starting and ending points (denoted t_s and t_e) of each step:

```
m1-put-on-robot$(k, c, c', r, d, p)$
   task: put-on-robot$(c, r)$
    pre: cargo$(r)$ = nil, top$(p) = c$, at$(p, d)$,
         attached$(k, d)$, holding$(k)$ = nil
    sub: (t1, navigate$(r, d)$),             // compound task
         (t2, unstack$(k, c, c', p, d)$),     // action
         (t3, load$(k, c, r, d)$)             // action
    stn: $t1_e < t3_s, t2_e < t3_s, 5 \le t1_s - t2_s \le 10$
```

Note that this STN is dynamically controllable □

The input to the learner is as in the basic case: a set A of durative actions, a set $\mathcal{T}$ of annotated tasks, and a temporal planner. However, we may consider temporally richer task annotations: both pre(τ) and eff(τ) can be temporally extended states with constraints (as in Section 18.5). The learner may rely on the same approach as before: (i) generate random temporal planning problems for the tasks in $\mathcal{T}$; (ii) generate temporal plans for these problems; (iii) synthesize a set $\mathcal{M}$ of methods from these plans; (iv) check the learned methods on new problems, extend $\mathcal{M}$ if needed.

The generated temporal planning problems will not be as rich as the chronicles of Section 17.1.4: each problem is focused on a single task τ with extended initial states (supported assertions) in pre(τ), assertions to be achieved as in eff(τ) and the constraints expressed in the annotation of τ. A planner such as TemPlan would generate for each problem a temporal plan as a set of primitive actions from A together with a dynamically controllable STN, if such a plan exists.

The synthesis of methods from temporal problems and plans may rely on an extended basic HTN learning procedure. Recall that the latter has three main steps:

- *Factorization*: Find a *contiguous* subsequence of steps in a given plan π of a task τ that matches a plan of another task τ'. In π, replace this subsequence by τ' as a subtask of τ. Repeat until no further subsequence replacements are possible.[1]
- *Lifting*: Replace constants with variables in the resulting methods.
- *Subsumption*: Remove subsumed lifted methods.

In principle, the lifting and subsumption procedures for the basic case should also work for the extended case, with a caveat regarding how STN subsumption is defined. We may consider that an STN network n_1 subsumes n_2 if they have the same set of nodes and if every instance that meets the constraints of n_2 meets also the constraints of n_1. However, keeping the most general method, i.e., the one with the least constraining network, might not be desirable in some cases. Alternate definitions should be considered, possibly on the basis of the partial order entailed from an STN, as explained next.

Changes required in the factorization procedure are substantial. We no longer consider contiguous subsequences of a sequence of steps but an STN embedding relationship.

[1]In this step, we are using the term "plan" loosely to mean a sequence of actions and tasks.

Let *Training* be the set of temporal planning problems and plans from which we want to synthesize $\mathcal{M}$. An element $(s_0, \tau, \pi) \in \mathit{Training}$ is such that s_0 is a temporally extended instance of pre(τ) and the plan $\pi = (A_\pi, \mathit{STN}_\pi)$ is the set of durative actions in π and the associated STN. Let $(s'_0, \tau', \pi') \in \mathit{Training}$ be another temporal plan for a task τ', with $\pi' = (A_{\pi'}, \mathit{STN}_{\pi'})$, such that:

- t'_s and t'_e are two nodes either existent or to be added to $\mathit{STN}_{\pi'}$ such that no node in $\mathit{STN}_{\pi'}$ precedes t'_s and no node is after t'_e; let us augment $\mathit{STN}_{\pi'}$ with these two nodes and edges to make them the starting and ending points of π'.
- $\mathit{STN}_\pi \setminus \mathit{STN}_{\pi'}$ is a network in which we remove from STN_π all nodes and edges of $\mathit{STN}_{\pi'}$ and replace all edges to and from $\mathit{STN}_{\pi'}$ to edges to t'_s or from t'_e.

A method m for τ from plan π can take τ' as a subtask in m if and only if $A_{\pi'} \subset A_\pi$ and $(\mathit{STN}_\pi \setminus \mathit{STN}_{\pi'}) \cup \mathit{STN}_{\pi'}$ is an equivalent network to STN_π.

Checking the latter condition on numerous pairs in *Training* is quite complex. It would probably make the learning procedure impractical. A possible simplification is to entail from each STN_π a partial order $\prec_\pi$ on A_π. In $\prec_\pi$ we drop from STN_π synchronisation and numerical constraints and keep only precedence constraints between actions in A_π. The embedding relation $(\prec_\pi \setminus \prec_{\pi'}) \cup \prec_{\pi'}$, simplifying the previous one on STNs, is computationally simpler. It may rely on preprocessing each partial order into the recognizer automata of its subsequences.[2] A final processing or even hand tuning of the learned methods may retrieve from *Training* some the needed temporal constraints.

The approach suggested here requires further investigation since it has not been, to our knowledge, developed and tested.

19.3 Discussion and Bibliographic Notes

Learning durative actions. In [444], a temporal PDDL domain from traces is learned by using grammar induction of classical plans, considered as regular languages, equivalent to Deterministic Finite Automata. Given a problem and a plan, a simulator samples a set of (good and faulty) time stamped plan instances. Action concurrency is allowed under the Single Hard Envelope (SHE) assumption: a durative action may concurrently run only with another durative action (called the envelope) which totally extends over it. The uses a two steps: an atemporal model is first learned then temporal information is added afterwards.

A based Constraint Satisfaction (CSP) approach is proposed in [389] to learn action specification from time-stamped traces. The work deals with causal-link relationships, condition threats and effect interferences. Grounded action and propositions pairs are associated with CSP variables and constraints with preconditions at start, duration, and end time. Given this representation, a general CSP solver is used. A solution of the CSP problem is a domain model consistent with observed traces.

[2]Note that "*bde*" or "*aca*" are subsequences of "*abcdae*" but not it substrings. A preprocessing in $O(n^2)$ builds the subsequence recognizer automata, which runs linearly for recognition.

Learning heuristics. Rather surprisingly, there has not been much work on learning heuristics for temporal planning. The only paper we are aware of is [790], which proposes a domain-independent learning and planning framework that, given a planning domain and a set of training problems, synthesizes a temporal planning heuristic for problems in the same domain. Reinforcement learning is used to construct a value function represented as a neural network, which is mathematically converted to a planning heuristic. In the proposed framework, a planning domain is defined with predicates, durative actions with minimal and maximal duration, add effects, delete effects and numeric effects as usual. In order to learn an approximation of the optimal heuristic, the learning problem is formalized as a model-free RL problem on an MDP encompassing the search spaces of the training instances. This encoding MDP represents a set of planning instances to be solved, and the optimal value function for the MDP is transformed into the optimal heuristic for all the planning problems. The input to the neural network encoding uses a fixed-size vector representation of the state of the MDP corresponding to a planning search state (assignment to the fluents, running actions and time representation) and the planning problem objective (the goal and the constants of the problem). The idea is to train the neural network to approximate the optimal value function for the MDP, aiming at generalizing to problems drawn from the same distribution as the training set.

Learning Chronicles. The chronicle representation is useful for planning as well as for monitoring and diagnosis. Learning chronicles for the latter case takes as input annotated logs of time stamp events. The learning approach of [404] relies on easily computed automata of subsequences and their Cartesian product, together with a clustering method based on longest common subsequences. Other contributions to chronicle learning such as [261, 1118] extend this approach with e.g., frequency information or additional knowledge. A variant of STN embeddings, called continuous-time dynamic network embeddings, have also been used for learning chronicles [692]. Some of these techniques can be relevant for learning extended temporal methods.

Temporal HTN. A few approaches have developed temporal extensions of HTN, e.g., as in the SIADEX planner [209]. HDDL2.1 is proposal for a temporal extension of the HDDL langage [881], where HTN methods are specified with an STN, as in the extended case of Section 19.2.2. Learning temporal HTN models has not received much attention. Note however an interesting application of SIADEX which synthesizes HTN domain models for educational learning from an adequate labelling of the learning domain repository [210].

Further works based on Neural Nets. Handling spatio-temporal data with neural nets is a widely studied issues, but only a few contributions consider explicit temporal variables and constraints. Specific functions to code and solve symbolic temporal constraints with a neural net have been proposed in [610]. Temporal graph neural nets have considered in several applications, in particular in social recommendation systems [76], but in most cases time is simply a sequence of graphs, possibly with some constraints [339].

19.4 Exercises

19.1. Explain the challenge of extending the algorithms for online learning of action schema in the deterministic case (see Algorithm 4.12) to take into account time.

19.2. Extend Algorithm 4.7 and Algorithm 4.8 presented in Section 13.2 to take into account durative actions.

19.3. Extend Algorithm 19.1 to deal with concurrent overlapping actions.

Part VII

Motion and Manipulation Models in Robotics

This movement was perfected after arranging, calculating and repeated trials.
Ismail al-Jazari, *The Book of Knowledge of Ingenious Mechanical Devices*, circa 1106

Changes in the world are triggered by movements. Acting requires moving. Almost all our everyday activities are based on motion and/or manipulation actions. Except for communication and passive sensing, very few tasks can be achieved without movements. The same holds for robots and embodied system that perceives, acts in and reasons about the physical world. This part of the book is about acting, planning and learning with motion and manipulation.

Planning in AI often focuses on the abstract parts of actions, leaving out their motion/manipulation parts. In several applications, such as logistics (until the recent development of logistic robots), actions are carried out by human drivers, pilots and workers whose admirable dexterity and motion capabilities should not be constrained nor interfered with. In these applications, movements do not need to be planned for.

Planning in robotics often focuses solely on motion and manipulation, leaving out the task and goal oriented part of planning. Concerns in robotics have been focused on their programming, allowing robots to find autonomously the adequate movements for a task, while the task planning is devoted to the human programmer or user.

Robotics and AI have matured enough to allow for handling complex applications, requiring autonomy in a diversity of tasks and changing environments. This requirement demands for the combined capabilities of reasoning on abstract actions as well as on concrete motion and manipulation steps. In the robotics literature, this is referred to as '*task aware planning*', i.e., planning beyond motion and manipulation. In the AI literature, it is referred to as '*combined task and motion planning*' (TAMP).

This class of TAMP problems, which includes task, motion and manipulation planning, is the topic of this part. The challenge in TAMP is the integration of symbolic models for task planning to metric models for motion and manipulation.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

Chapter 20 introduces the representations and techniques for achieving and controlling motion, navigation, and manipulation actions in robotics. Chapter 21 discusses motion and manipulation planning algorithms, and their integration with task planning in TAMP problems. Chapter 22 covers learning for the combined task and motion-manipulation problems. We generally assume here deterministic and fully observable domains. We review in the discussion sections techniques that handle the uncertainty of sensors and actuators.

20 Motion and Manipulation Actions

This chapter sets the foundation for the next two chapters. It introduces the reader to the use of robotics platforms for the development of acting, planning and learning functions. The mechanical design of robot platforms is a topic beyond our scope, for which additional reading is given in the discussion.

The study of motion is deeply rooted in physics and astronomy. It goes back to Archimedes. Newton, Leibniz, Lagrange, and Euler set the basis of classical mechanics for the modeling of forces and their effects on mouvements. Motion is also the topic of other disciplines, such as biomechanics to dynamic control. Robotics builds up on this broad knowledge to master computational motion, navigation, and manipulation over different types of devices and environments.

Robotic devices are informally introduced in the following section. Motion problems and the *metric representations* with continuous state variables needed for geometric, kinematic and dynamic operational models are then presented. Section 20.3 introduces localization and navigation problems, followed by a section on manipulation problems and their representations. The chapter ends with a general discussion and a few exercises.

20.1 Robots

A robot is a machine able to perform a set of concrete *tasks* in a class of *environments* by perceiving, moving, manipulating and interacting *physically* with its environment.[1] A robot integrates several types of components, among which:

- *actuators*, e.g., motors, hydraulic linear actuators, artificial muscles; actuators transform energy into forces and elementary motions;
- *effectors*, e.g., joints, grippers, wheels, legs, wings; effectors use actuators to move the robot in the environment and manipulate objects;
- *sensors*: the *proprioceptive* ones estimate the robot state, e.g., odometer, inertial measurement unit, GPS; the *exteroceptive* sensors estimate the environment, e.g., camera, laser, radar, lidar, sonar, infrared, tactile and haptic sensors;
- computation and communication devices;
- sources of energy, e.g., battery, fuel cell.

There are several types of robots corresponding to different classes of applications, environments and tasks. Well-known examples are:

- *Manufacturing robots*: arms with adapted sensors attached to fixed positions for tasks such as painting, welding, assembly, loading and unloading a press or a machine tool;

[1]Note that this definition excludes abstract machines such as chatbots.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- *Service robots*: mobile platforms in indoor environments for cleaning, surveillance, transportation in a shop, a workshop, a warehouse, or a hospital;
- *Exploration robots*: ground vehicles in outdoor environments for mapping, soil analysis, mining, intervention and exploration in remote areas or planets;
- *Aerial and underwater drones*, for exploration and intervention tasks from the air or in the sea;
- *Personal robots*: mobile robots assisting people in professional environments or at home;
- *Medical robots*: robots specialized in assisting surgeons, e.g., in "noninvasive" or high precision surgery.

Other types of robots, such as exoskeletons, that restore or extend the sensory-motor capability of a person, or robots for agriculture, construction, demining or military operations give rise to specific developments. Teams of robots or human and robots bring additional challenges for interaction and cooperation.

A robot performs its tasks through a *perception–decision–action* loop. For that, it may be endowed with some level of *cognitive autonomy*, i.e., a capability to grasp what it senses and decide on the basis of its perceptions how best to undertake and pursue its task. The autonomy in energy, tools and other resources may constrain the task achievement.

A key issue is the *diversity of environments and tasks* a robot must face, and how flexible and versatile it must be. When there is no diversity, i.e., for a single task or a single environment type, a robot can benefit from extended prior modeling and engineering. It does not need much cognitive autonomy. Numerous successful applications have been deployed, e.g., robots in the manufacturing industry, vacuum cleaners, lawn mowers, or autonomous ground logistic vehicles used in warehouses, hospitals, or electronic cleanrooms.

Diversity of environments and tasks requires either autonomous robots or a *human in the loop* to drive the robot. Tele-operated robots benefit from human perception and deliberation functions. They have been deployed in complex applications, e.g., surgery or planet exploration. Challenge are how to provide comprehensive sensory feedback to a remote human operator to enable her to properly understand the state of the environment and the task, and how to reliably translate human commands to the robot actuators (*e.g.*, give a haptic feedback to a surgeon and filter the signal from the movements of her hands to obtain a precise and safe trajectory of the scalpel and control its motion with respect to the movement of the operated organ).

Tele-operation, even when desirable, constrains the tasks that can be performed. The early Mars rovers, *Spirit* and *Opportunity*, were initially tele-operated at the motor control level; communication delays (up to 40 minutes) limited their remote operation to a few meters per day. *Curiosity*, then *Perseverance* rovers benefited from more autonomous motion; they are still tele-operated at the exploration task level. When autonomy is not desired, it is often preferable to tele-operate a robot at the task level, *e.g.*, order it to make a precise line of surgical sutures, or to close an underwater valve, but leave it up to the robot to translate the task into precise motion and controlled commands, under the supervision of the operator. Here also, the state of the art has reached some maturity, illustrated for example by robots used in hazardous environments. Telepresence robots are another illustration of task-level tele-operation. These are mobile platforms carrying away the image and voice of a user, giving her a visual and audio feedback, capable of simple tasks, e.g., find a person, asking for an object and bringing it back.

The cognitive autonomy of a robot becomes critical when the diversity of environments and tasks is important, precluding extensive prior engineering for all the tasks, and when tele-operation is too costly or not feasible (e.g., for a fleet of coordinated drones). However, cognitive autonomy is not a binary property, either true or false. The more a robot needs to be versatile and adaptive to its missions, the more it needs to be autonomous. For example, autonomy at the task level would allow a robot to perform precisely defined tasks in its usual range of environments. Autonomy at the mission level would permit a robot to be given a mission in general terms, at a high level of abstraction, possibly with a utility function, e.g., find and rescue injured persons in the area. The mission takes place in an open environment, possibly unknown and unusual to the robot, and may involve a collection of tasks. Human interactions, as in personal robots, add specific uncertainty, and variability constraints in the environments and tasks.

Autonomy at the mission level is easier when the environment is variable, but the tasks are well structured and constrained. Driverless cars are a good illustration. Autonomous underwater vehicles are another example, e.g., experimental AUVs have been launched for a full day mission for mapping, sampling, oceanographic and biological measurement.

Robotics research relies significantly on experiments. The advance of the field has been conditioned by the availability of inexpensive reliable platforms with broad functionalities that are easily deployable and programmable. Significant progress has been witnessed in the last decade. A good illustration is provided by humanoid robots. Many research groups experiment with complex biped robotic platforms of human size, e.g., the '*Pyrène*' robot in Figure 20.1. These robots demonstrate good motor skills as well as impressive mechatronics. Platforms on wheels with two arms, sometimes with an articulated trunk, also illustrate rich sensory-motor capabilities. Platforms such as the '*Justin*' robot (Figure 20.6(a)) are able for example to catch simultaneously two balls thrown from few meters, to fold laundry or open doors.

Several research competitions stimulated the progress of the field. In addition to autonomous driverless cars, there are several other competitions, e.g., in robotics assembly, aerial robotics or humanoid robotics. The popular "*RoboCup*" competition has several tracks, e.g., logistics with interesting workshop servicing tasks. Some competitions started with oversimplified "micro-worlds" problems. However, their effects in attractiveness, visibility, team commitment, and progress measurement remain largely beneficial to the progress of robotics.

20.2 Motion

To control its movements, a robot needs to represent the shape of its environment and of its limbs, and, for the latter, to represent how they are articulated, can move and what forces they can exert. Representing shapes is the topics of *geometry*; motion is

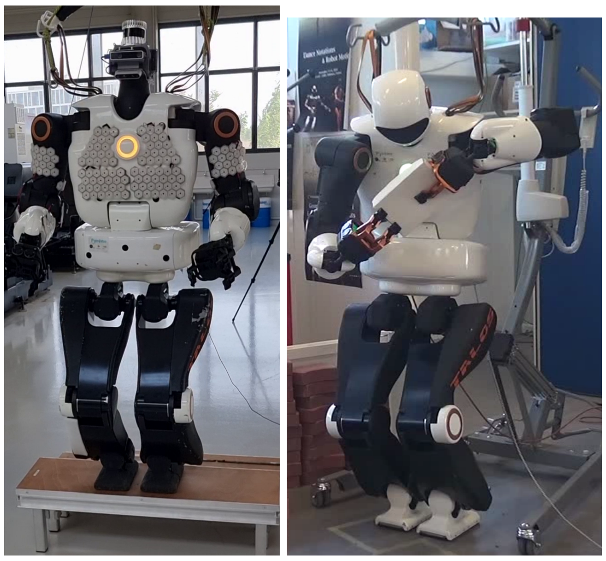

(a) (b)

Figure 20.1. The '*Pyrène*' robot (a) climbing stairs and (b) performing a manipulation task with coordinated whole body motion planning [1054, 850].

described by *kinematics*, forces by *dynamics*. This section introduces the reader to the basics of these three areas. For simplicity, we assume a static environment and focus on the motion of the robot limbs and the movable objects it is in contact with.[2]

20.2.1 Geometry

Shapes are modeled with a geometric description of the following sets:

- $\mathcal{W}$ is the set of fixed and movable parts of a domain, i.e., walls, boundaries, pieces of furniture, obstacles and objects. Let $\mathcal{O} \subseteq \mathcal{W}$ be the set of movable parts of the domain, i.e., objects and tools of interest to the task that can be grasped, pushed, and moved.
- $\mathfrak{R}$ is the set of limbs and effectors (e.g., grippers, hands) of the robot performing the task.

These sets are composed of three-dimensional parts described with CAD geometric models in $\mathbb{R}^3$.

Convex polyhedra. A CAD model of rigid parts relies on the specification of convex polyhedra. A polyhedron is described with a finite set of vertices, edges and faces. Each vertex is a point in $\mathbb{R}^3$ located in space with respect to a Cartesian reference frame $\mathcal{F}$. An edge is an ordered pair of vertices. A face is a list of vertices, ordered counterclockwise with respect to the outside normal to the face (vector $\vec{n}$ in

[2]In human-robot interaction and other domains, the robot motion has to take into account the movements of others.

Figure 20.2), together with the linear equation of the plane of this face in a Cartesian frame. An edge as (v_2, v_3) appears in the adjacent face in the opposite order.

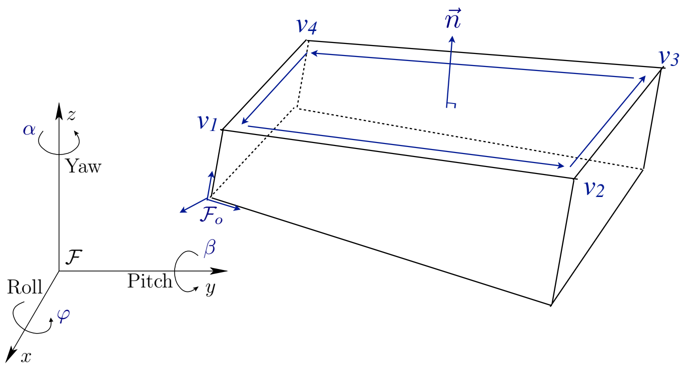


Figure 20.2. A simple convex polyhedron. The vertices, ordered edges and normal to the upper face are marked; a local Cartesian frame $\mathcal{F}_o$ attached to the polyhedron and a global frame are shown.

Let $f_i(x, y, z) = a_i x + b_i y + c_i z + d_i$ be the equation in a frame $\mathcal{F}$ of a planar face i, i.e., $f_i(x, y, z) = 0$ for the points of the face and $f_i(x, y, z) > 0$ for any point in space above the face with respect to the normal to this face. A convex polyhedron o defined with a set of faces f_i, for $i \in I_o$, is the solid corresponding to the set of 3D points

$$o = \{(x, y, z) \text{ in } \mathcal{F} \mid \forall i \in I_o, f_i(x, y, z) \leq 0\}. \tag{20.1}$$

A non-convex polyhedron can be defined as the union of a finite set of possibly overlapping convex polyhedra. The decomposition of a non convex polyhedron into convex ones is not unique. It allows approximating any complex shape with cavities and holes (with a caveat for the sense of the normals of polyhedral holes). Other primitives than the polyhedron can also be used if needed in more complex CAD models, e.g., cylinders, spheres, curved surfaces with splines and quadrics. Note that the geometry of a domain may also rely partly on bitmap models. These decompose with some resolution a surface or a volume into a 2D or a 3D grid, each cell of which being associated with a scalar v about the content of the cell. For example, in a numerical model of an outdoor terrain, v gives the elevation of each cell, in an occupancy grid, v may approximate the probability of having an obstacle in a cell. Bitmaps can be conveniently acquired from a robot range sensors.

Pose in space. A rigid object o defined with a polyhedron can be precisely positioned in the 3D Euclidian space with a vector of 6 parameters $\mathbf{q} = [x, y, z, \alpha, \beta, \varphi]$, called the *pose* of o in space. The pose vector gives the coordinates in a global Cartesian frame $\mathcal{F}$ of a reference point in o, and three angles, called roll, pitch and yaw, which define the orientation of o in $\mathcal{F}$. These 6 parameters are called the *degrees of freedom* (*dof*) of o in space (see Figure 20.2).

It is convenient to associate with an object o a local Cartesian frame $\mathcal{F}_o$ attached to o in a reference point. The pose vector $\mathbf{q} = [x, y, z, \alpha, \beta, \varphi]$ can be defined as the position and orientation of the local frame $\mathcal{F}_o$ in the global frame $\mathcal{F}$. Any point in o is positioned in the local frame $\mathcal{F}_o$ given the geometric model of o. The coordinates of this point in a global $\mathcal{F}$ are computed from its coordinate in $\mathcal{F}_o$ and its pose $\mathbf{q}$ with simple rotation and translation operations as illustrated next.

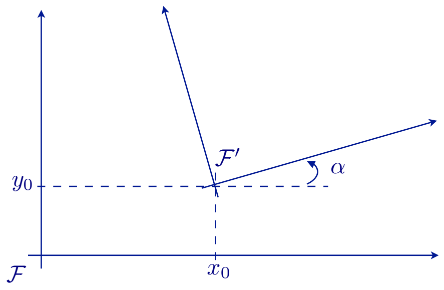


Figure 20.3. Geometric positioning of a 2D frame

Example 20.1. Consider for simplicity, a 2D local Cartesian frame $\mathcal{F}'$ located with respect to a global frame $\mathcal{F}$ with a translation of its origine (x_0, y_0) and a rotation α in the trigonometric sense (Figure 20.3). In 2D, the pose of $\mathcal{F}'$ with respect to $\mathcal{F}$ is defined with a vector $\mathbf{q} = [x_0, y_0, \alpha]$. A point of coordinates $[x', y']$ in $\mathcal{F}'$ has the coordinates $[x, y]$ in $\mathcal{F}$ that are computed as follow:

$$\begin{aligned} x &= x_0 + x' \cos\alpha - y' \sin\alpha \\ y &= y_0 + x' \sin\alpha + y' \cos\alpha \end{aligned}$$

In matrix notation:

$$[x, y]^\top = \rho_\alpha \times [x', y']^\top + [x_0, y_0]^\top, \text{for } \rho_\alpha = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}.$$

If $\mathcal{F}'$ is attached to a polygonal object o moving in $\mathcal{F}$, every vertex, edge and point of o is fixed and located in $\mathcal{F}'$ with the equation of o. It can be positioned in $\mathcal{F}$ with the above relation. □

Relative positioning and scene graphs. A local frame $\mathcal{F}_o$ as associated with a polyhedra o. The pose of o in a global frame $\mathcal{F}$ is specified with the 6 parameters $\mathbf{q} = [x_0, y_0, z_0, \alpha, \beta, \varphi]$, which correspond to a translation vector $[x_0, y_0, z_0]^\top$ for the position in $\mathcal{F}$ of the origine point of $\mathcal{F}_o$, and three rotation matrices ρ_φ, ρ_β and ρ_α (see Appendix B). These matrices define respectively the roll, pitch and yaw rotations of $\mathcal{F}_o$ with respect to the x, y and z axes of $\mathcal{F}$ (Figure 20.2). The mapping of a point from coordinates $[x', y', z']^\top$ in $\mathcal{F}_o$ to coordinates in $\mathcal{F}$ is given as:

$$\begin{aligned} [x, y, z]^\top &= \rho_\alpha \times \rho_\beta \times \rho_\varphi \times [x', y', z']^\top + [x_0, y_0, z_0]^\top \\ &= \mathcal{H}_\mathbf{q}([x', y', z']) \end{aligned} \qquad (20.2)$$

where $\mathcal{H}_\mathbf{q}$ is the global transformation from $\mathcal{F}_o$ to $\mathcal{F}$ when o is in the pose $\mathbf{q}$.[3]

[3] $\mathcal{H}_\mathbf{q}$ can be defined as a matrix in the '*homogeneous transformation*' notation.

In summary, every object in the environment $\mathcal{W}$ is specified with 3D CAD primitives and located in $\mathcal{F}$. A local Cartesian frame $\mathcal{F}_o$ is located in $\mathcal{F}$ with the transformation $\mathcal{H}_\mathbf{q}$ associated with a pose. An object pose changes over time and updated given the achieved and/or observed motions.

The relations between local and global frames can be organized and maintained hierarchically, e.g., frames for objects in a shelf or on a table, for shelves in a cabinet, for cabinets and tables in a room, etc. This can be done with a convenient and widely-used data structure, called a *scene graph*.

A scene graph is tree which relates the objects in a scene and allows tracing their relative positions. To each object corresponds a node in the tree with a triple $(o, parent, \mathcal{F}_o)$:

- o is the CAD geometric description of the object
- *parent* is a pointer link to the parent node
- $\mathcal{F}_o$ is the local reference frame of the object with respect to its parent node.

The root node provides the global reference frame of the environment. The scene graph can also refer to fixed areas in the environment, e.g., rooms, corridors, etc.

The description of geometric shapes of objects can be extended with other properties of interest to the tasks, such as physical properties of objects, e.g., color, weight, friction parameters for manipulation, or semantic and functional properties, e.g., types of objects (cups, plate, doors, tools). Note that some objects may be articulated, e.g., a door (with two mobile parts, a panel and a knob) or a drawer; they require a kinematic description (see Section 20.2.2). Furthermore, it is desirable to link to geometric description of $\mathcal{W}$ to the topology of the environment, e.g., a map of a building telling which room is connected to which other places.

Localization and collision testing. The global data structure description of a domain is designed for the efficient computation of frequent operations needed for motion and manipulation actions, mainly:

- *localization*: where is an element $o \in \mathcal{W}$, and what are its geometric relations to other parts of the domain, e.g., which face of o is resting on which face of which other element of the domain;
- *collision testing*: is a point in free space, or is it within or close to some object of $\mathcal{W}$ and which.

The latter is a highly frequent operation in motion and manipulation planning. Its efficient implementation relies on scene graph and additional data structures, e.g., octrees or a hierarchy of bounding boxes such as to reduce the testing of Equation 20.1 from a linear complexity in the number of faces to a logarithmic one.

The representation needs to handle the bounded precision of the models and the inaccuracy of the robot sensors and actuators, e.g., the uncertainty for grasping an object on a table has to take into account the inaccuracy of the table and object models and that of the robot perception system that located the object's pose on the table. Note also that the representation should allow maintaining consistency constraints, e.g., objects cannot float in space or rest on less than three points. More complex balance constraints may need to be maintained, e.g., instable or narrow faces, tilted supports.

To sum up, geometric models describe the *task space* of the robot, in which it moves, which is the Euclidian 3D space (sometime constrained to 2D for a robot moving on a plane), associated with global and local Cartesian frames, with adequate geometric transformation matrices between them. In addition to these geometric descriptions, we need to specify how a robot can move, how to compute its motion, and what are the associated constraints. This is the topic of kinematics.

20.2.2 Kinematics

Geometry describes the shapes of individual objects and parts and specifies the task space of the robot. Kinematics considers movements of articulated chains of bodies. Given the geometric description of the parts in an articulated chain, as well as the specification of the articulation links between the parts, and the movement of one or a few parts in the chain, kinematics allows computing the mouvements and positions of the other parts. A robot moves because its articulations are actuated. The movement is controlled through these actuations. A main issue is to relate the task space to actuation space.

Joints and degrees of freedom. The limbs or parts of a robot are connected through articulations, called *joints*, that can be of different types. Two parts P_1 and P_2 in an articulated chain can be connected mainly through:

- *a revolute joint*: P_2 rotates around a rotation axis fixed in a P_1, e.g., the hinges or the handle of a door;
- *a prismatic joint*: P_2 slides along a translation axis fixed in a P_1, e.g., a drawer in a cabinet.

Recall that a solid free in space has 6 degrees of freedom (*dof*). A part linked in a kinematic chain will loose some of its *dof* because of the constraining joint. For example, a part connected with a revolute joint will have just one *dof*: the rotation angle around the joint axis. Similarly a prismatic joint leaves only one *dof* along the translation axis. The position of $P2$ with respect to $P1$ depends on how they are connected with a joint j, and on the current value, denoted θ_j, of the rotation or translation along the joint axis. This scalar value θ_j is called the current *configuration* of the joint j. Given the position of $P1$ in a global Cartesian frame and the configuration θ_j of the joint with $P2$, the position of any point in $P2$ can be computed using a transformation $\mathcal{H}_{\theta_j}$ (Equation 20.2).

This computation can be carried out transitively through a kinematic chain, i.e., from $P1$ to $P2$ then $P3$, etc. It can be simplified if a local Cartesian frame is adequately positioned with respect to the joint axis, e.g., align axis z_o with the axis of a revolute joint.

To sum up, a joint j is specified with its type, the position of its axis in the two connected parts, its current angular or linear configuration θ_j, and the limited amplitude of the movement, i.e., the maximum and minimum values of θ_j.

The revolute and prismatic joints have a single axis. There are however other types of joints that have several axes. They can be seen as the composition of two or more revolute and prismatic joints. These are for example:

- the *cylindrical joint* that has two axes, a rotation and a translation;
- the *universal joint*, with two rotation axes;
- the *planar joint* with three axes: two translations and one rotation within a plane;
- the *spherical joint* with three rotation axes corresponding to the roll, pitch and yaw angles.

Let us take an intuitive example: the human skeleton. Many of our joints are spherical (e.g., shoulder, elbow, hip), some are universal (wrist), a few are revolute joints (last joints of the fingers); we do not have prismatic joints.[4]

A joint allows for as many *dof* as its number of axes. A *screw* is a particular single-axis joint although it allows for both a rotation and a translation, but these two movements are coupled; there can be a single translation value given a rotation and vice versa, hence a single *dof*.

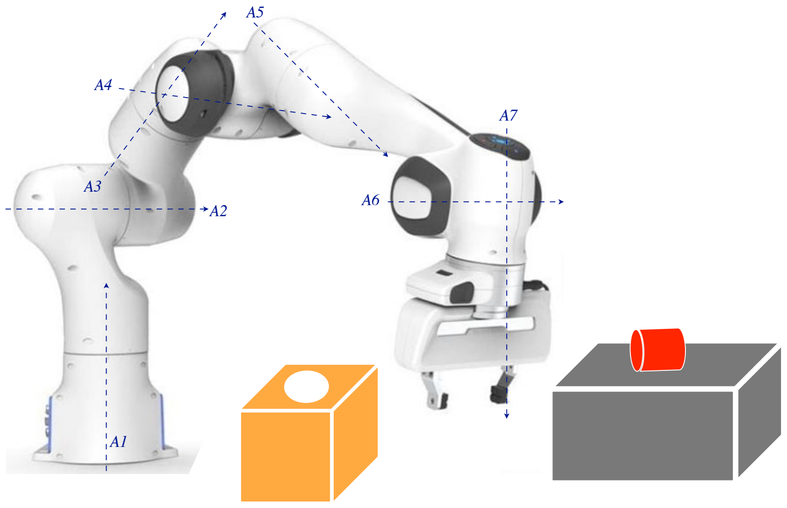


Figure 20.4. The Franca Emika robot (©2024 Franka Robotics GmbH). This arm has 7 *dof*. Its first component is fixed to a base and linked serially to the other components with 7 revolute joints around the axes labelled *A1* to *A7* in the figure. The gripper has two fingers with one *dof* prismatic joint.

The number n of *dof* of an unconstrained kinematic chain is the sum of the degrees of its joints.[5] A robot arm has usually 6 or 7 *dof* (e.g., Figure 20.4), not counting those of the hand, which may have up to 20 degrees for a five-finger hand (two revolute and one universal joints per finger). The Justin robot Figure 20.6(a) has 53 *dof* (7 per

[4]But we have some stretching capability, since our body is not composed of rigid links.
[5]Kinematic constraints can reduce n.

arm, 12 per hand, plus the base, the torso and the head).[6] A complex biped humanoid robot may have a hundred *dof*. The *dof* of a robot may be structured as a sequence of links, as for a robot arm, or as a tree of links, as for a humanoid robots that has several limbs, or event possibly as a graph with closed kinematic chains, e.g., two arms holding the same object form a closed chain.

Let $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_n] \in \mathbb{R}^n$ be the current configuration vector of the joints of a robot. These n *dof* may be constrained, i.e., it may not be possible to change a configuration parameter θ_i independently of others. For example, a car has three *dof*; it can reach any position and orientation in a plane free of obstacles (like for a planar joint), however a car cannot move sideway. This is called a *holonomy* constraint.[7] A non-holonomous robot may require elaborate movements to move between two configurations. Consider for example the car maneuvers required to perform a parallel parking in a tight space.

In a robot arm, the joints are serially connected. For a biped robot, the joints are connected as a tree with several branches, for the legs, arms, torso and head, plus those corresponding to the hands. In some robots, the kinematic connexions correspond to a cyclic graph (chains with loops). These are for example Stewart platforms, which consist of two plates connected with up to 6 prismatic joints. A robot carrying a tray with two arms maintain a cyclic chain. Cyclic kinematic links entails additional movement constraints.

In a robot arm, every *dof* is usually actuated, e.g., in Figure 20.4 a motor is associated with each of the 7 revolute joints. However, some of the *dof* of an n degrees robot may not be actuated. For an actuated degree j, the robot can apply a force to change θ_j. The movement of a non-actuated degree may be due to gravity forces, or to the movement of other degrees and kinematic constraints. This movement can be predicted but not directly actuated. For example, the early Hilare robot Figure 20.5 had a car-like kinematics, with three *dof*: a *differential drive*, i.e., two independently actuated drive wheels, and a front caster wheel which rotates freely along its z axis under the movement of the drive wheels. Note that although this robot moves in the 3D space, it has the kinematics of a rigid body on a plane (see Figure 20.5(b)), it's task space is 2D (informally referred to as a 2.5D).

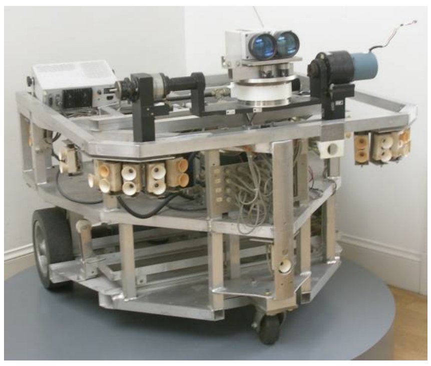
Configuration space. Let $C \subset \mathbb{R}^n$ be the set of possible values of the configuration vector $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_n]$ of a robot $\mathfrak{R}$ with n *dof*. C takes into account the limited amplitude of each degree θ_j and the kinematic constraints. It is called the *configuration space* of a robot.[8] Let $\mathfrak{R}^{\boldsymbol{\theta}}$ be the space occupied by the robot when in configuration $\boldsymbol{\theta}$, i.e., the subset of $\mathbb{R}^3$ in the frame $\mathcal{F}$ occupied by $\mathfrak{R}$. The robot *end-effector*, i.e., its hand or gripper, is of specific interest for handling object. Let $\mathbf{q}_{\mathit{eff}}(\boldsymbol{\theta})$ be the pose in $\mathcal{F}$ of this end-effector when the robot is in configuration $\boldsymbol{\theta}$.

$\mathfrak{R}^{\boldsymbol{\theta}}$ and $\mathbf{q}_{\mathit{eff}}(\boldsymbol{\theta})$ are computed from the geometry of each robot part, and from the

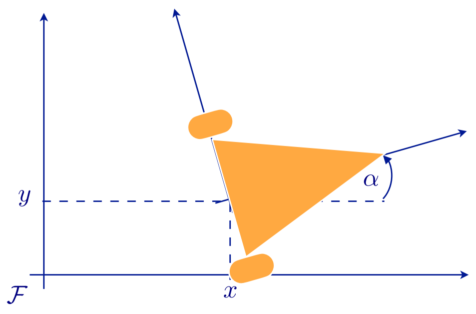
[6] See https://www.dlr.de/en/rm/research/robotic-systems/humanoids/agile-justin

[7] A kinematic equivalent of the constraints seen earlier between state variables defining a state space S.

[8] As defined, this is more precisely the *joint space*. But the two are identical for serial or tree-structured kinematics. In a closed kinematic chain, constraints entail less configuration parameters then joint parameters.



(a)



(b)

Figure 20.5. (a) The mobile Hilare robot developed at LAAS in the seventies, now exhibited at the CNAM museum in Paris. It has a differential drive, i.e., two independently actuated drive wheels and a front free wheel. (b) Simplified schema of Hilare's kinematics with the (x, y, α) parameters.

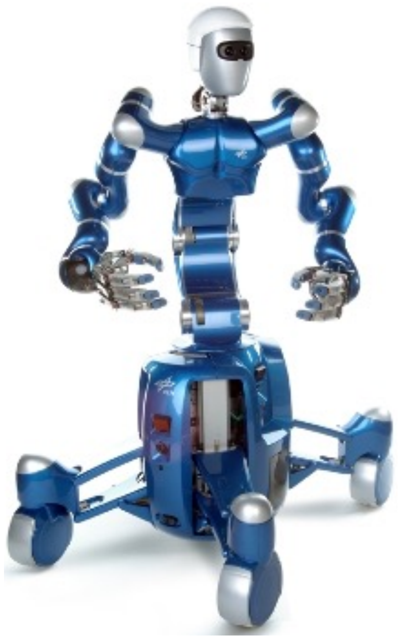
transformations $\mathcal{H}_{\theta_j}$ positioning part $(j + 1)$ with respect to part j when the joint configuration is θ_j.

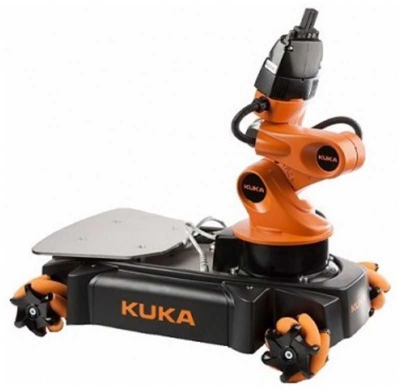
The movements of a robot are computed and controlled in the configuration space C. Consider for example the movement required to put the arm in Figure 20.4 in a position allowing it to close its gripper for grasping the red peg. The robot will have to raise its hand, and move it such as to have the gripper axis orthogonal to and centered with respect to the red peg axis. This needs to be done while not getting in touch with the obstacles, including the red cylinder. This movement is performed with appropriate rotations around the axes *A1* to *A7* , i.e., in the configuration space. Finding the appropriate movements can be formalized as finding a path in the configuration space C from an initial configuration $\boldsymbol{\theta}$ to a final one $\boldsymbol{\theta}'$ that avoids the obstacles.

The movements of a robot are actuated and controlled in its configuration space C, but its task (e.g., the red cylinder to be grasped) and its obstacles are in the task space, i.e., $\mathbb{R}^3$. How do we map the task space, in which the robot acts, into the configuration space in which it moves C, and vice versa?

Forward and inverse problems. To answer the above question, let us first address two simpler dual problems which consider a robot alone in space, ignoring for the moment possible obstacles and collision constraints:

- *the forward kinematic problem* is to compute a mapping from the configuration to the task spaces, i.e., from C to $\mathcal{F}$: given a robot configuration $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_n]$, what is the pose $\mathbf{q}_{\mathit{eff}}(\boldsymbol{\theta}) = [x, y, z, \alpha, \beta, \varphi]$ of its end-effector (or the pose of another part of interest of the robot) in the task space relative to the frame $\mathcal{F}$;
- *the inverse kinematic problem* is to compute a mapping from the task to the configuration spaces, i.e., from $\mathcal{F}$ to C: given a goal pose $\mathbf{q}_g = [x, y, z, \alpha, \beta, \varphi]$


(a)



(b)

Figure 20.6. (a) The Justin robot of the DLR that has 53 *dof* [167]; (b) The YouBot, a small omnidirectional platform with an arm totaling 8 *dof*, predecessor of today's mobile platforms of Kuka AG.

in the task space relative to $\mathcal{F}$, is there a configuration $\boldsymbol{\theta}$ which puts the robot end-effector (or another part of interest in $\mathfrak{R}$) in position $\mathbf{q}_{eff}(\boldsymbol{\theta}) = \mathbf{q}_g$.

The forward problem can be stated with n equations and 6 unknowns. It is solved from the geometric and kinematic models by propagating the transformation equations 20.2 from one joint to the next. These computations are conveniently structured with a scene graph (p. 447), extended with nodes for the robot's joints and links. The pose of each join is computed from the local frame of the robot and its configuration $\boldsymbol{\theta}$.

The inverse problem may or may not have a solution. It requires solving a system of 6 nonlinear equations with n variables, usually with numeric approximation methods. If the goal pose $\mathbf{q}_g$ is within reach of the robot, there is one solution for $n = 6$ and infinitely many for $n > 6$. The latter case corresponds to a robot that is *kinematically redundant*, that is, it has more degrees of freedom than strictly needed, and can use a preference criterion to choose a particular solution to the inverse problem.

The full specification of the geometric and kinematic models of a robot can be a quite tedious task. Fortunately, this task is now greatly simplified thanks to a standard *Universal Robot Description Format* (URDF) based on XML. Commercially available robots come with their URDF computational models and a specification of their reachability space (see Figure 20.7). These models use scene graphs. They can be run for solving the forward and inverse problems with several efficient software tools (some are included the robotics middleware ROS[9]).

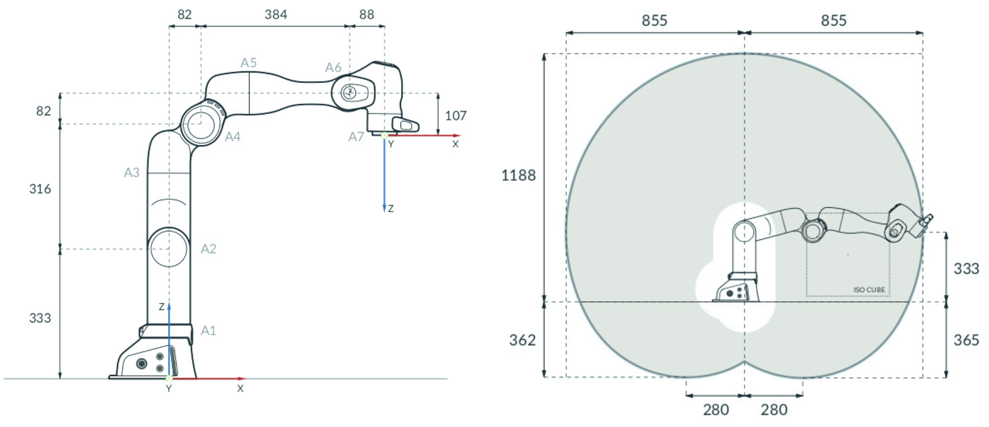


Figure 20.7. Part of the geometric specification and reachability workspace (side view) of the Emika Franca robot (from the Franka Datasheet document).

The problem of mapping obstacles and the geometry of the environment from the task space to the configuration space is unfortunately much more complex.

Example 20.2. Consider a planar robot arm that has just two *dof* with a circular obstacle in its reachable space (Figure 20.8(a)). The projection of this simple 2D obstacle in the simple 2D configuration space has an irregular shape (red area in Figure 20.8(b)). It can be constructed by finding the set of configurations $[\theta_1, \theta_2]$ which keep the arm in contact with the obstacle. If the amplitude of the first joint is limited to $-\pi \leq \theta_1 \leq +\pi$, then this obstacle divides C into two subspaces: from A the robot cannot reach point B. Moving the obstacle away from the base would open a path from A to B. □

Collision detection and distance to obstacles. In an environment with obstacles, a robot movement should be actuated and controlled in the part of C not occupied by obstacles. Let C_{obs} be the set of configurations in which the robot is in contact or collision with obstacles, i.e., $C_{obs} = \{\boldsymbol{\theta} \in C | \mathfrak{R}^{\boldsymbol{\theta}} \cap \mathcal{W} \neq \varnothing\}$. The *free configuration space*, denoted C_{free}, is $C_{free} = C \backslash C_{obs}$. Finding C_{obs} requires mapping the obstacles from the task space to the n-dimensional configuration space. The computation of such a mapping is based on the idea of finding all possible contacts between a limb of $\mathfrak{R}^{\boldsymbol{\theta}}$ and an obstacle in $\mathcal{W}$. Even when all bodies in $\mathfrak{R}$ and $\mathcal{W}$ are convex polyhedra, the procedure for computing the projection of $\mathcal{W}$ into the robot configuration space is very complex.

Fortunately, it is easier to test whether or not the robot $\mathfrak{R}^{\boldsymbol{\theta}}$ in a configuration $\boldsymbol{\theta}$ is in collision with, or at some distance from an object $o \in \mathcal{W}$, when o is in a pose $\mathbf{q}$.

[9]See the ROS tutorial on URDF: http://wiki.ros.org/urdf/Tutorials

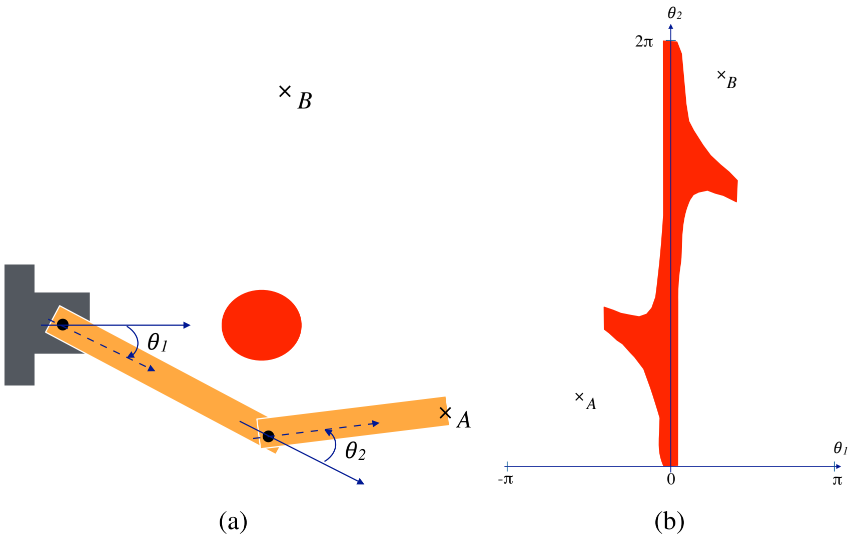


Figure 20.8. (a) A planar robot with two angular joints, θ_1 and θ_2, facing a red circular obstacle. (b) The configuration space of this robot: the projection of the obstacle in C shows that the two configuration $\boldsymbol{\theta}_A$ and $\boldsymbol{\theta}_B$ are not connected : if $-\pi \leq \theta_1 \leq +\pi$ no possible motion takes the robot from points A to B.

Let o^q be the set of points in the task space occupied by o when in pose $\mathbf{q}$, and let

$$\delta(\mathfrak{R}^{\boldsymbol{\theta}}, o^{\mathbf{q}}) = \begin{cases} -\infty & \text{if } \mathfrak{R}^{\boldsymbol{\theta}} \cap o^{\mathbf{q}} \neq \varnothing \\ min_{\boldsymbol{x}\in\mathfrak{R}^{\boldsymbol{\theta}},\ \boldsymbol{x}'\in o^{\mathbf{q}}} \|\boldsymbol{x} - \boldsymbol{x}'\|^2 & \text{otherwise} \end{cases} \tag{20.3}$$

where $\boldsymbol{x}$ and $\boldsymbol{x}'$ are the vector coordinate in the task space of two points in respectively $\mathfrak{R}^{\boldsymbol{\theta}}$ and o, and $\|\boldsymbol{x} - \boldsymbol{x}'\|$ is Euclidean distance between $\boldsymbol{x}$ and $\boldsymbol{x}'$. The value of $\delta(\mathfrak{R}^{\boldsymbol{\theta}}, o^q)$ indicates a collision or gives the closest distance obstacle o of the robot when in configuration $\boldsymbol{\theta}$.

The distance δ can be efficiently computed on the basis of a hierarchical decomposition of $\mathfrak{R}^{\boldsymbol{\theta}}$ and o as *trees of bounding boxes*, as in octrees. A bounding box is an imaginary volume, e.g., a sphere, a cube, or a rectangular prism defined with three intervals over the three axis of a frame $\mathcal{F}$. The root of the tree is a global box that includes all of the a robot. It is decomposed into boxes for the limbs, which are decomposed into boxes for the limb parts, etc. Similarly for the trees of boxes bounding objects. If two root boxes dot not intersect, there can be no collision, otherwise we'll have to check the descendent boxes.

Bounding boxes are chosen such as to be easily computed from the geometry of a polyhedra (e.g., as with rectangular prisms), and such that the collision test and distance computation between two boxes b and b' are computed in $0(1)$. A procedure for computing δ starts from the boxes associated with the roots of the two trees associated with $\mathfrak{R}$ and o. If the two root boxes overlap in space the procedure is refined on their subtrees. At some pair of boxes, if $\delta(b, b') > \eta$, for η an *error margin* parameter, the procedure stops on this refinement branch. Otherwise it compares their respective child boxes. The procedure exits when a collision is detected on a branch. At each level, hashing techniques can filter out collision free cases. An incremental version of the procedure caches the results of previous collision tests given the planed movements of the robot.

To sum up, a robot has to switch back and forth from its task space, where it moves and where its obstacles and objects of interest lay, to its configuration space where its motion paths are defined, where it is actuated and controlled. This mapping from task to configuration spaces is difficult to do globally but is easily computable for each given point.

Path in the free configuration space. A robot motion from an initial configuration $\boldsymbol{\theta}_0$ to a goal configuration $\boldsymbol{\theta}_g$, both in C_{free}, corresponds to a *continuous path* in the free configuration space defined as a function $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]} : [0,1] \rightarrow C_{free}$ such that $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(0) = \boldsymbol{\theta}_0$ and $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(1) = \boldsymbol{\theta}_g$.

Definition 20.3. A path in the configuration space between two configurations $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$, both in C_{free}, is a continuous function $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}$ which maps a real $\zeta \in [0,1] \subset \mathbb{R}$ into free configurations, such that $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(0) = \boldsymbol{\theta}_0$ and $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(1) = \boldsymbol{\theta}_g$. □

The parameter ζ is the curvilinear coordinate along the path and $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(\zeta)$ is the robot configuration at that point. In the simplest case, a path may defined as a line *segment* in the configuration space:

$$\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(\zeta) = \boldsymbol{\theta}_0 + \zeta(\boldsymbol{\theta}_g - \boldsymbol{\theta}_0), \quad \text{for } \zeta \text{ varying continuously from 0 to 1.} \tag{20.4}$$

Note that a segment in the configuration space is *not* a straight line in the task space. For example, a line segment for the planar robot in Figure 20.8 from $[-\pi/4, \pi/4]^\top$ to $[-\pi/4, -\pi/4]^\top$ is a circular arc around θ_2.

C is a connected manifold, i.e., there exist a path between any pair of points in C.[10] However, because of obstacles, C_{free} may have several connected components (see Example 20.2). Paths exist only within each component, but not between them. Hence, a collision-free path in C_{free} exists only when $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$ are in the same connected component. In the latter case, there will be a collision free path, but not necessarily a segment path in C_{free}. For example, if the obstacle in Figure 20.8 is slightly moved away from the base, there will be a free path from A to B (e.g., rotate θ_2 towards $-\pi/2$, then θ_1 towards $\pi/2$, then increase θ_2 while reducing θ_1), but not a line segment in C_{free}. We would need a more complex function than Equation 20.4 to define a path, e.g., a sequence of line segments.

A significant advantage of the configuration space representation is to conceptually simplify the motion of a complex articulated body in space as a continuous path of the movement of a *point* $\boldsymbol{\theta}$ in the free configuration space.

Assuming that a path $\wp$ exists and has been computed (see Chapter 21), how can we move the robot along $\wp$? The simplest way would be to actuate the robot such as to follow the configurations given by the path $\wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_g]}(\zeta)$, when varying ζ continuously

[10]A manifold is a space which is locally similar to (homeomorphic) the Euclidian space.

from 0 to 1. Since by definition $\wp_{[\theta_0,\theta_g]}(\zeta) \in C_{free}$ for $0 \le \zeta \le 1$, there can be no collision along this path in C_{free}. However, such a movement in C_{free} does not account for the speed and acceleration of the robot along $\wp$, nor for the commands needed to be given to the actuators of the robot in order to achieve this movement, for their limitations, constraints, and the uncertainty on their effects. These issues are discussed in the next section.

20.2.3 Dynamics

Kinematics provides a collision free path $\wp \in C_{free}$ which ignores time as well as the masses, inertia, forces, torques and balance constraints of the robot. A movement that takes into account these constraints is called a "*kinodynamic motion*". The path description $\wp$ is extended with information about the dynamics. A *trajectory* is a *velocity function* along a path $\wp$ which is feasible by the robot, i.e., a function meeting its actuators and inertia constraints. A robust execution of a trajectory is controlled with a feedback information from the robot sensors. Hence we need to find a trajectory as well as a *feedback control law* to track this trajectory. To do so, dynamic models integrate geometry and kinematics together with the masses, moment of inertia and friction parameters of the robot components. Dynamics allow computing the forces needed to move a robot along a path $\wp$ while keeping it balanced (e.g., as in Figure 20.1) and to determine its resulting acceleration and speed. It relies on Newton classical mechanics and its computational developments.

The material needed to adequately cover dynamic models and control of robots goes beyond the scope of this section. We give here an intuition of the main concepts and problems involved in the design and use of a robot platform. Let us start with simple issues.

Transition function. The movement of a robot can be considered as a state transition system. In the discrete deterministic case we denoted $s' = \gamma(s, a)$ as the next state reached from a state s with an action a. The equivalent of s is here a configuration state $\boldsymbol{\theta} \in C$. The state variables are simply the components of vector $\boldsymbol{\theta}$. The equivalent of a is here a *vector actuation command* $u \in \mathcal{U}$, whose components are the joint actuation variables in a continuous action space. The state transition function for the movement is written as:

$$\dot{\boldsymbol{\theta}} = \gamma(\boldsymbol{\theta}, u), \text{ where } \dot{\boldsymbol{\theta}} = \frac{d\boldsymbol{\theta}}{dt} \text{ is the velocity vector of } \mathfrak{R}. \tag{20.5}$$

If $\mathfrak{R}$ has n *dof*, Equation 20.5 corresponds to n scalar equations, that is γ is an n vector function.

Example 20.4. Consider the Hilare robot in Figure 20.5(b). It has 3 *dof*: $\boldsymbol{\theta} = [x, y, \alpha]$. It is actuated with two independent motors. Let us take $u = [u_l, u_r]$, the rotation velocities of the left and right wheels (in radians per second). Let ρ be the radius of the wheels, and ν be the width of the robot, i.e., the distance between its wheels. The three components of the state transition function γ for this robot are:

$$\dot{x} = \frac{\rho}{2}(u_l + u_r)\cos\alpha, \quad \dot{y} = \frac{\rho}{2}(u_l + u_r)\sin\alpha, \quad \dot{\alpha} = \frac{\rho}{\nu}(u_r - u_l)$$

When $u_r = u_l$ the robot translates along a straight line. When $u_r = -u_l$ the robot rotates around the center of its local frame, which remains fixed. Its rotation speed is proportional to the wheel radius ρ, and inversely proportional to wheels distance ν. Note that each component of γ is a function of both u_l and u_r. However, we can redefine the actuation space as $u = (u_t, u_o)$, with $u_t = (u_l + u_r)/2$ and $u_o = u_r - u_l$:

$$\dot{x} = \rho u_t \cos\alpha, \quad \dot{y} = \rho u_t \sin\alpha, \quad \dot{\alpha} = \frac{\rho}{\nu} u_o.$$

In this actuation space, u_t and u_o respectively translate and rotate the robot. □

Jacobian of a robot. An adequate model relating the actuation u, the acceleration $\ddot{\boldsymbol{\theta}}$, speed $\dot{\boldsymbol{\theta}}$, and configuration $\boldsymbol{\theta}$, allows computing the velocity of the robot end-effector (or of any other part of interest in $\mathfrak{R}$) in the task space. Symmetrically, a required velocity function of the end-effector needs to be mapped to the adequate actuation in the configuration space. These two symmetric issues correspond to the already mentioned forward and inverse kinematic problems, considered here at the dynamic level. The forward problem is expressed as the following equation:

$$\dot{\mathbf{q}}_{\mathit{eff}}(\boldsymbol{\theta}) = J(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{20.6}$$

where $\dot{\mathbf{q}}_{\mathit{eff}}(\boldsymbol{\theta}) = [\dot{x}, \dot{y}, \dot{z}, \dot{\alpha}, \dot{\beta}, \dot{\varphi}]^\top$ is the velocity vector of the end-effector in the task space, that is the derivative of the already seen pose $\mathbf{q}_{\mathit{eff}}(\boldsymbol{\theta})$, and $J(\boldsymbol{\theta})$ is the *Jacobian* matrix of the robot. J is a matrix of dimension $6 \times n$, whose terms are functions of $\boldsymbol{\theta}$, defined as $J(\boldsymbol{\theta}) = \left[\partial v_i / \partial \boldsymbol{\theta}_j\right]$, where v_i is the ith component of v, for $1 \le i \le 6$, and $\boldsymbol{\theta}_j$ the jth component of $\boldsymbol{\theta}$ for $1 \le j \le n$.

Example 20.5. Consider the simple planar robot of Example 20.2 and let us extend it with a third revolute joint and a third component, giving a 3 *dof* configuration space: $\boldsymbol{\theta} = [\theta_1, \theta_2, \theta_3]$. Let l_1, l_2 and l_3 be the respective distances between the three joints. This robot being planar, the position of its end-effector in the task space is defined with three parameters: (x, y, α). The forward kinematic model of this simple robot is:

$$\begin{aligned}
x &= l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3) \\
y &= l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2) + l_3 \sin(\theta_1 + \theta_2 + \theta_3) \\
\alpha &= \theta_1 + \theta_2 + \theta_3
\end{aligned}$$

The Jacobian matrix of the robot is: $J(\boldsymbol{\theta}) = \begin{bmatrix} \partial x/\partial\theta_1 & \partial x/\partial\theta_2 & \partial x/\partial\theta_3 \\ \partial y/\partial\theta_1 & \partial y/\partial\theta_2 & \partial y/\partial\theta_3 \\ \partial\alpha/\partial\theta_1 & \partial\alpha/\partial\theta_2 & \partial\alpha/\partial\theta_3 \end{bmatrix}$

The partial derivative terms are easily computed, e.g.,

$$\begin{aligned}
\partial x/\partial\theta_1 &= -l_1 \sin\theta_1 - l_2 \sin(\theta_1 + \theta_2) - l_3 \sin(\theta_1 + \theta_2 + \theta_3) \\
\partial y/\partial\theta_2 &= l_2 \cos(\theta_1 + \theta_2) + l_3 \cos(\theta_1 + \theta_2 + \theta_3) \\
\partial\alpha/\partial\theta_1 &= \partial\alpha/\partial\theta_2 = \partial\alpha/\partial\theta_3 = 1
\end{aligned}$$

□

The same method applies for computing the Jacobian of a more complex robot, as the 7 *dof* robot of Figure 20.4. In the 3D task space, the partial derivatives for the (x, y, z) position parameters are usually easily computed, while it can be technically tedious to find the partial derivatives for the (α, β, φ) angular parameters.

Singularities of a robot. From the statement of the forward problem with Equation 20.6, we can formulate the inverse problem as:

$$\dot{\boldsymbol{\theta}} = J^{-1}(\boldsymbol{\theta})\dot{\mathbf{q}}_{\mathit{eff}}(\boldsymbol{\theta}) \tag{20.7}$$

where J^{-1} is the inverse of the Jacobian, when defined, or the pseudo-inverse matrix. Recall that the inverse problem may or may not have one or an infinite number of solutions. When J is a square matrix, i.e., when $n = 6$ for a 3D task space (or $n = 3$ in the 2D case as in Example 20.5), $J(\boldsymbol{\theta})$ is invertible, that is $J^{-1}(\boldsymbol{\theta})$ is well defined, iff determinant$(J(\boldsymbol{\theta})) \neq 0$. In that case there is a single solution to the inverse problem, solved with Equation 20.7.

When determinant$(J(\boldsymbol{\theta})) = 0$, the matrix $J(\boldsymbol{\theta})$ is said to be *singular*; there is no solution to the inverse problem. The values of $\boldsymbol{\theta}$ for which $J(\boldsymbol{\theta})$ is singular are the *singularity configurations* of a robot. In a singularity, the robot loses one or several *dof*, its motion is restricted. It is important to plan the motion of a robot away from obstacles, as well as from its singularities. For most manipulators and simple robots (see Exercise 20.4 for the robot in Example 20.5), these singularities can be computed analytically in advance and taken care of at a low-level control of the robot.

For a redundant robot (i.e., $n > 6$), Equation 20.7 uses a pseudo-inverse Jacobian (more precisely a Moore–Penrose inverse) computed by solving numerically an optimization problem for some criteria, e.g., minimizing energy.

The main relations seen so far in this section, i.e., $\dot{\boldsymbol{\theta}} = \gamma(\boldsymbol{\theta}, u)$ and $\dot{\mathbf{q}}_{\mathit{eff}}(\boldsymbol{\theta}) = J(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}$, are about velocity, but do not deal with dynamics per se, i.e., forces, acceleration, masses and inertia. This is done with the following ordinary differential equation:

$$\tau = M(\boldsymbol{\theta})\ddot{\boldsymbol{\theta}} + C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})\dot{\boldsymbol{\theta}} + g(\boldsymbol{\theta}) \tag{20.8}$$

Here τ is the torque vector in the configuration space. The first term in the right hand side corresponds to the well-known Newton's second law equation (a force is a mass times acceleration); here the matrix $M(\boldsymbol{\theta})$ corresponds to the apparent masses of the links in a given configuration. The second term corresponds to the angular momentum and Corriolis forces as a function of the configuration and velocity. The last term is the gravitational forces for the current configuration.

For a slow moving robot, not handling objects and tools, Equation 20.8 is often ignored or simplified. But for fast motions and complex interactions with the environment, it has to be developed to know the forces and torques needed at the joints to move the robot. This equation accounts only for a dynamic motion. When exerting forces on the environment, a fourth torque term needs to be added in Equation 20.8.

Consider for example a robot handling a drill, a polishing or other tools requiring a pushing force. What are the additional torques at the joints needed to produce the force vector $f_{\boldsymbol{\theta}}(eff)$ at its end-effector? This question corresponds to the forward dynamic problem addressed again with the Jacobian matrix:[11]

$$\tau = J(\boldsymbol{\theta})^{\top} f_{\boldsymbol{\theta}}(eff) \tag{20.9}$$

The inverse problem, i.e., a mapping from the joint torques to the end-effector forces, raises the already discussed issues of inverse or pseudo-inverse of the Jacobian.

Generally, motor actuations correspond to torques and require complicated computations with second order derivatives from Equation 20.8. However, it is often convenient to express an ordinary differential equation with second or higher order terms as a set of equations with only first order terms, as illustrated next.

Example 20.6. Let us go back to Example 20.4 and consider that a torque command on a motor does not set directly it speed but accelerates it. Hence the actuations u_l and u_r accelerate the left and right motors, requiring an integration to get the left and right motor speeds, denoted ω_l and ω_r. This gives the following second order model:

$$\begin{aligned} &\dot{\omega}_l = u_l, \qquad\qquad \dot{\omega}_r = u_r,\\ &\dot{x} = \frac{\rho}{2}(\omega_l + \omega_r)\cos\alpha, \quad \dot{y} = \frac{\rho}{2}(\omega_l + \omega_r)\sin\alpha, \quad \dot{\alpha} = \frac{\rho}{\nu}(\omega_r - \omega_l). \end{aligned}$$

This model needs to be completed with inertia and friction forces, that brake the robot and require a traction force to keep constant the speed. □

20.2.4 Trajectory and Control

Recall the distinction between paths and trajectories. A path $\wp$ is a continuous function in the free configuration space ignoring time, while a trajectory is a velocity function along $\wp$ meeting the actuation and dynamic constraints of the robot. How do we refine a path into a trajectory?

Time scaling. A convenient method for mapping a path into a trajectory relies on a *time scaling function*. Instead of taking a scalar $\zeta \in [0,1]$ as a coordinate argument of $\wp$, let us define ζ as a function of time: $\zeta : [0,T] \rightarrow [0,1]$, where T is the total time given to the robot to perform the movement. We may have an apriori given constraint to be met by the scaling function, i.e., $T \leq T_{max}$, i.e., reach the final point of the path in less than T_{max}.

For simplicity, let us denote $\boldsymbol{\theta}(\zeta(t))$ the robot configuration at time t on the path $\wp$. With these notations, the velocity and acceleration functions are:

$$\dot{\boldsymbol{\theta}} = \frac{d\boldsymbol{\theta}}{d\zeta}\dot{\zeta}, \quad \ddot{\boldsymbol{\theta}} = \frac{\dot{\boldsymbol{\theta}}}{d\zeta}\ddot{\zeta} + \frac{d^2\boldsymbol{\theta}}{d\zeta^2}\dot{\zeta}^2$$

To apply the above equations, the time scaling function ζ and $\boldsymbol{\theta}(\zeta)$ need to be differentiable. In simple cases, the idea is to accelerate the robot as much as possible

[11] The power exerted by the robot is identical at its joints and at its end-effector, i.e., $\dot{\boldsymbol{\theta}}^{\top}\tau = \nu^{\top} f$. Equation 20.6 gives ν as a function of $\dot{\boldsymbol{\theta}}$, leading to: $\dot{\boldsymbol{\theta}}^{\top}\tau = (J\dot{\boldsymbol{\theta}})^{\top} f = \dot{\boldsymbol{\theta}}^{\top} J^{\top} f$, hence $\tau = J^{\top} f$.

to a maximum speed, keep that speed constant in a cruse phase, then decelerate it to end $\wp$ at a zero speed. Angular momentums may require deceleration in part of the path, as in sharp turns for cars.

Example 20.7. A very simple ζ function would start with a high impulse acceleration $\ddot{\zeta}$ at $t = 0$ then decelerate linearly until T. This would give a parabola for $\dot{\zeta}$ and a third order polynomial for ζ. The four parameters of this polynomial are determined with the four constraints: $\zeta(0) = 0, \zeta(1) = T, \dot{\zeta}(0) = 0, \dot{\zeta}(T) = 0$ (see Exercise 20.5). □

The drawback of the above simple time scaling function is the initial high impulse acceleration: even when feasible and sufficient for moving along the path $\wp$, it is quite agressive for motors. Higher order polynomials allow for smoother accelerations. S-curve time scaling interpolates continuously between accelerated, constant speed, and decelerated phases. Finally, B-spline functions allow for smooth trajectories defined with *control points* along $\wp$, requiring the robot to stay within a convex hull close to these configuration points, but not constraining it to pass through them.

Motion and force control. We are now able to define a *desired* trajectory for performing a movement. But we have to expect differences between what is desired and what is achieved. These differences are due to imperfect models and inaccurate sensors and actuators. Control theory is used to track and reduce these differences as much as possible through feedback mechanisms.

Motion control is the basic case if we want to solely control the motion of the robot between two configurations, without doing anything. *Force control* seeks to control the forces exerted by its end-effector. *Hybrid control* is the interesting case where we need to control both the forces and the motion. Consider the example of a robot opening a door. Here we need forces as well as motion for grasping and turning the knob and for pushing the door while getting in. Similarly for a robot erasing a board: the pushing forces have to be controlled along with the robot motion.

Let us first focus on the basic case of motion control. From a planned path σ and a trajectory specified with a scaling function ζ, we can define the desired configuration and velocity functions, denoted $\boldsymbol{\theta}_d$ and $\dot{\boldsymbol{\theta}}_d$. Let $\boldsymbol{\theta}_e = \boldsymbol{\theta}_d - \boldsymbol{\theta}$ be the *feedback error* or discrepancy between the desired and measured configurations. The current configuration $\boldsymbol{\theta}$ is usually measured with odometers, motor optical encoders and displacement transducers. Angular rate sensors and gyroscopes can also be used to estimate $\boldsymbol{\theta}$ by integration.[12]

Let us first assume that the actuation variable u allows setting the desired velocity $\dot{\boldsymbol{\theta}}_d$ (as in Example 20.4).

An *open control* actuates the robot by taking the transition function (Equation 20.5) simply as $\dot{\boldsymbol{\theta}} = \dot{\boldsymbol{\theta}}_d$. An open loop control is a kind of blind movement that can end up way off the path and be dangerous for the robot's environment.

A *closed loop* control uses corrective terms as a function of the feedback error $\boldsymbol{\theta}_e$ (Figure 20.9). In the simple case of a proportional controller, the actuation input is:

[12]The derivative of a signal is very noisy; hence position measures are not good for estimating $\dot{\boldsymbol{\theta}}$. Integration of a signal is quite robust but accumulates drift over time.

$$\dot{\boldsymbol{\theta}} = \dot{\boldsymbol{\theta}}_d + K\boldsymbol{\theta}_e, \text{ where } K = I_{(n,n)}k, \text{ for } I \text{ identity matrix.}$$

When the scalar $k > 0$, this controller allows $\boldsymbol{\theta}$ to track the desired configuration $\boldsymbol{\theta}_d$. The larger k is, the faster $\boldsymbol{\theta}$ goes to $\boldsymbol{\theta}_d$.[13]. But a too large k exhausts the actuators, which limit possible values.

This simple controller leaves a non-null remaining error, called the steady state (or asymptotic) error. A proportional integrative (PI) controller gets rid of this error with an additional term, computed from $\boldsymbol{\theta}_e$:

$$\dot{\boldsymbol{\theta}} = \dot{\boldsymbol{\theta}}_d + K\boldsymbol{\theta}_e + K' \int_0^t \boldsymbol{\theta}_e dt$$

Here too K' is a diagonal matrix: $K' = I_{(n,n)}k'$; we require k and k' to be positive for a stable tracking with no steady state error. A too large k' leads to overshooting of $\boldsymbol{\theta}$ with respect to $\boldsymbol{\theta}_d$, amortized with possible oscillations. A stability analysis shows that a fast tracking response without overshooting is given with $k' = k^2/4$.

Note that this control takes place in the configuration space. It can however be transposed to the task space, which is important in particular to minimize the error for the pose of the end-effector.

Let us now consider the general case where the actuation sets the joint forces and torques (as in Example 20.6). Let us assume that the robot motion is not too fast such as to neglect the inertial and dynamic forces. The controlled torque for moving the robot will be a function of $\boldsymbol{\theta}_e$ and $\dot{\boldsymbol{\theta}}_e$, with $\dot{\boldsymbol{\theta}}_e = \dot{\boldsymbol{\theta}}_d - \dot{\boldsymbol{\theta}}$. The desired $\dot{\boldsymbol{\theta}}_d$ is computed from the scaling function, the measured $\dot{\boldsymbol{\theta}}$ is given by a speed sensor. A proportional integrative derivative (PID) controller actuates the robot according to the following equation:

$$\tau = K\boldsymbol{\theta}_e + K' \int_0^t \boldsymbol{\theta}_e dt + K''\dot{\boldsymbol{\theta}}_e.$$

This popular PID controller is easy to tune and has generally good stability properties.

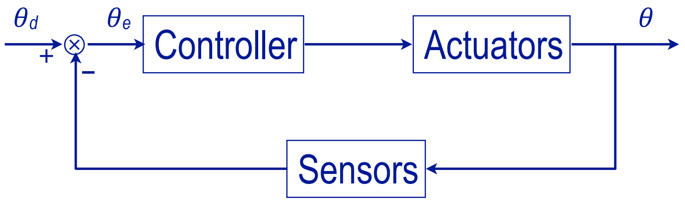


Figure 20.9. A simple closed-loop feedback control diagram with respect to a desired trajectory. The output of the control is for example $\dot{\boldsymbol{\theta}}$ or τ given by the PI or PID controllers.

If dynamic forces cannot be neglected, then Equation 20.8 has to be taken into account. This is also the case for a force or hybrid controllers, where we end up

[13] K is the inverse of the tracking time constant.

controlling a torque combining Equation 20.8 and Equation 20.8:

$$\tau = M(\boldsymbol{\theta})\ddot{\boldsymbol{\theta}} + C(\boldsymbol{\theta}, \dot{\boldsymbol{\theta}})\dot{\boldsymbol{\theta}} + g(\boldsymbol{\theta}) + J(\boldsymbol{\theta})^\top f_{\boldsymbol{\theta}}(\mathit{eff})$$

The controller uses extension of the above methods. It relies on $\boldsymbol{\theta}_e, \dot{\boldsymbol{\theta}}_e$, $f_e = f_d - f$ and the desired $\ddot{\boldsymbol{\theta}}_d$, as well as force and torque sensors to measure the exerted force $f_{\boldsymbol{\theta}}(\mathit{eff})$. More elaborate *model predictive controllers* rely on optimization techniques to handle constraints and multi-objectives trad-offs.

20.2.5 Potential Fields

The previous approaches require a pre-planned path from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$, as well as a trajectory and a control law to actuate the robot along the chosen path. Instead of that, the idea here is to define a *potential function* over C_{free} which entails a field of forces that drives the robot toward $\boldsymbol{\theta}_g$ and away from obstacles. Imagine something akin to the gravitational potential of a planet which drives the movement of a free particle in space. To parallel what was seen in previous parts of the book, the path related approaches are similar to acting with a sequential plan, while here the potential function defines a *continuous policy* over the entire space. As any other policy, the potential corresponds to a closed loop control.

A potential field is a differentiable function $\phi : C_{free} \to [0, +\infty[$. For a given goal $\boldsymbol{\theta}_g$ and environment, we can take ϕ as the sum of:

- ϕ_a, an attractive field towards $\boldsymbol{\theta}_g$, regardless of obstacles, and
- ϕ_r a repulsive field away from obstacles, regardless of the goal.

$\phi(\boldsymbol{\theta}) = \phi_a(\boldsymbol{\theta}) + \phi_r(\boldsymbol{\theta})$. The forces entailed by a potential field are defined as:

$$F(\boldsymbol{\theta}) = -\left[\frac{\partial \phi}{\partial \boldsymbol{\theta}_i}\right] = -\nabla\phi(\boldsymbol{\theta}) = -\nabla(\phi_a(\boldsymbol{\theta}) + \phi_r(\boldsymbol{\theta})) = F_a(\boldsymbol{\theta}) + F_r(\boldsymbol{\theta})$$

A simple attractive potential is $\phi_a(\boldsymbol{\theta}) = \|\boldsymbol{\theta} - \boldsymbol{\theta}_g\|^2$; hence $F_a(\boldsymbol{\theta}) = -2(\boldsymbol{\theta} - \boldsymbol{\theta}_g)$. This potential is a parabola whose minimum is $\boldsymbol{\theta}_g$. A control following F_a is simply a greedy gradient descent toward the minimum.

For the definition of a repulsive potential, let $o_{\boldsymbol{\theta}}$ be the closest obstacle to $\mathfrak{R}^{\boldsymbol{\theta}}$ and $\delta(\boldsymbol{\theta}, o_{\boldsymbol{\theta}})$ be the distance given by Equation 20.3. The repulsive potential can be defined as:

$$\phi_r(\boldsymbol{\theta}) = \begin{cases} 0 & \text{if } \delta(\boldsymbol{\theta}, o_{\boldsymbol{\theta}}) > \text{threshold} \\ 1/\delta(\boldsymbol{\theta}, o_{\boldsymbol{\theta}}) & \text{otherwise} \end{cases}$$

The repulsive force F_r is null if there is no nearby obstacle, otherwise $F_r = (\boldsymbol{\theta} - \boldsymbol{\theta}_o)/\delta(\boldsymbol{\theta}, o_q)^2$; $\boldsymbol{\theta}_o$ being the configuration matching the closest point in $\mathfrak{R}^{\boldsymbol{\theta}}$ to o. The closer $\mathfrak{R}$ is to o, the bigger is the repulsive force which grows quadratically with the inverse squared distance.

To sum up, a potential field is an *a priori* given continuous policy defined everywhere the field is: $\pi(\boldsymbol{\theta}) = -\nabla\phi(\boldsymbol{\theta})$. The potential field approach is appealing since in particular it is *reactive* and can be coupled to online sensing. The robot may not know in advance about some obstacles. It will sense, update its repulsive field locally and avoid obstacles in its way towards $\boldsymbol{\theta}_g$. This is feasible even when obstacles are moving without a prior model of their motion (e.g., human in the environment); the goal may also be a moving target. Attractive and repulsive terms are set with respect to perception feedback.

The potential field approaches are incomplete methods. They suffer from the usual drawback of local minima. The robot may get trapped into a local minimum; its alternative is to get momentarily farther from its goal, seeking randomly an open path, which it may or may not find. Furthermore, if $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$ are in two connected components of C_{free} (as in Example 20.2), a robot following a potential field policy may wander for a while before giving up, unable to distinguish between many local minima and the lack of solution.

Fortunately, it is possible to combine in a mixed approach a planed path and a potential field to remain reactive to new and moving obstacles.

Motion in discretized space. Another computational motion approach, which is quite popular in applications such as, e.g., computer games, relies on a grid discretized space. This approach uses graph search algorithms applied to grids, with a few adaptations such as domain specific heuristics. But a straightforward use of a grid discretization does not simplify much the mapping from the task space to the configuration space. Furthermore, the approach does not scale up to a highly dimensional configuration space.

However, it is possible to combine a hierarchical and irregular discretization of space with potential field methods, in particular for a mobile platform with arms (e.g., Figure 20.6).

The main idea is to focus the discretization on the 2D projection of the task space, which is also part of the configuration space for a mobile platform. The approach is schematically the following:

- decompose the 2D free space into a topological graph where a vertex is a convex region free of obstacles, an edge corresponds to two adjacent vertices in the free space,
- compute a global potential field on this graph, e.g., with a Dijkstra all-pairs shortest path algorithm,
- for a motion from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$ follow such a potential, but of the last vertex (where $\boldsymbol{\theta}_g$ is), which is to be refined with a planned path and/or an attractive/repulsive potential in order to reach the final robot arm pose.

The decomposition method conditions the quality of the approach. It can be defined hierarchically, e.g., ignore at a coarse level the orientation of the platform and focus only on the fixed part of the environment $\mathcal{W} \setminus \mathcal{O}$, ignoring movable objects in $\mathcal{O}$. The approach can also be applied to a flying drone with a hierarchical space decomposition into 3D regions, small ones at low altitudes and bigger ones in higher free space.

Finally, let us note that space decomposition approaches overlap with and are relevant to navigation problems, discussed next.

20.3 Navigation

A fixed robot arm moves in a well engineered and precisely modeled environment possibly equipped with sensors to sharply localize the end-effector in the task space. A moving robot faces a navigation problem, which takes place in the *task space*. Navigation problems are usually focused on the movements of a point-like platform (in 2D for a ground robot, or 3D for an aerial one), ignoring the detailed configuration parameters of arms and other limbs.[14]

Navigation abstracts away configuration parameters but considers motion at a broader level, in space and semantically. It also raises the issue of mapping (or updating a map) and localizing the robot within this map. This issue, classically referred to as Simultaneous Localization and Mapping (SLAM) is discussed next. Global positioning systems have simplified many localization problems. But GPS is not available everywhere (e.g., indoor or between high buildings), and often it is the relative not the global positioning which constrains the task. To operate autonomously in a diversity of environments, a mobile robot must be able to locate itself directly from a map of its environment and from the perceived elements of this environment. This map is often only partially known. Uncertainty is a central issue in navigation; localization error is to be estimated, when beyond a threshold, specific actions have to be done.

Navigation for mobile robots is addressed at the metric level as a simultaneous localization and mapping problem, which is developed next. We'll then discuss navigation at the topological and semantic levels.

20.3.1 Simultaneous Localization and Mapping

Navigation focuses on the position of the robot local frame defined with a vector $\boldsymbol{x}$, and its movement $\boldsymbol{u}$ resulting from a command (considered here as the movement vector, not as a velocity or a force). A major issue in navigation is to handle the sensing and actuation inaccuracy and the map uncertainty. In this context, the *simultaneous localization and mapping* (SLAM) problem corresponds to two tightly coupled subproblems:

- *Localization*: let us assume the robot to be initially localized in a known environment, modeled by k landmarks, considered to be static, easily recognizable and perfectly positioned points in the task space. At time t, the robot is in a position estimated by $\tilde{\boldsymbol{x}}_t$. It moves by $\boldsymbol{u}_t$. This allows estimating the new position $\tilde{\boldsymbol{x}}'$. The robot then observes the landmarks where it expects to find them given $\tilde{\boldsymbol{x}}'$. It updates its estimated position in relation to each recognized landmark. The observed positions of the landmarks are combined into a new estimated position of the robot $\tilde{\boldsymbol{x}}_{t+1}$. The process is repeated at each time step as long as the robot remains within a fully known environment. The intermediate estimate $\tilde{\boldsymbol{x}}'$ allows solely to find landmarks. The localization error takes into account the sensing errors in the observed landmark positions, but these errors

[14]Similarly, air navigation deals with just 3 configuration parameters (plus possibly time), while an airplane motion requires 6.

do not increase (while the odometer error does). Hence, the error associated with the movement $\boldsymbol{u}_t$ does not affect the localization.

- *Mapping*: The robot builds a map of its environment assuming it knows precisely its successive positions. The j^{th} landmark is estimated at time t as $\tilde{\boldsymbol{x}}_{j_t}$. The robot moves between t and $t+1$ to a new known position, from which it observes again the position of the j^{th} landmark as $\tilde{\boldsymbol{x}}'_j$ with a sensing error. $\tilde{\boldsymbol{x}}'_j$ and $\tilde{\boldsymbol{x}}_{j_t}$ are combined into a better estimate. The map quality improves over time (Figure 20.10).

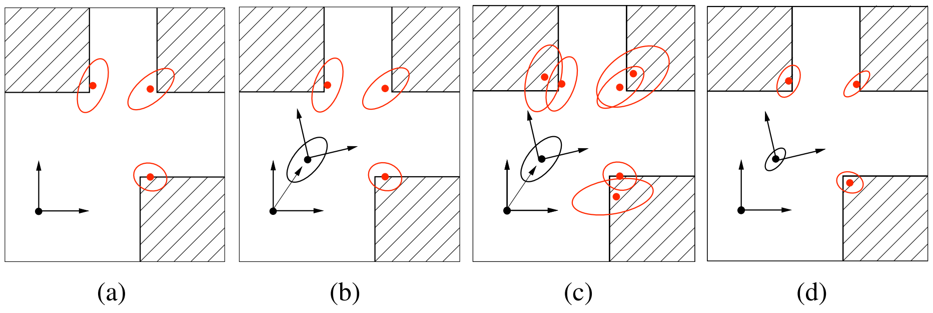


Figure 20.10. SLAM procedure for a simple 2D robot: (a) Three landmarks (corners of obstacles) are detected and positioned with some inaccuracy due to sensing noise (red ellipses). (b) The robot moves and estimates its position with a motion error (black ellipse). (c) The landmarks are observed and associated with the corresponding ones previously perceived. (d) Data fusion reduces the errors on the current position of the robot and the positions of the landmarks. The process is iterated for each new robot motion and sensing.

Localization assumes an error free map, while mapping assumes an error free motion. However, the initial map, if there is one, is never error free. Errors in the map entail localization errors. Symmetrically, the robot localization is noisy, which entails errors in its updates of the map. Fortunately, the sensing and motion sources of errors are *uncorrelated*. By addressing simultaneously localization and mapping, we combine the two subproblems into the simultaneous estimate of the positions of the robot and the landmarks, and reduce uncertainty on both estimates.

Kalman filtering SLAM. One approach for solving SLAM relies on *extended Kalman filters*. The technical details may seem complicated but a step by step presentation shows that the principle is simple. It is assumed that the environment is static and the sensors of the robot are properly calibrated and do not introduce a systematic bias. Sensing errors are modeled as a Gaussian noise with zero mean and a standard deviation σ specific to each sensor.

Consider the case of having two sensors, whose deviations are respectively by σ_1 and σ_2, which both measure the distance to the *same* landmark. They return two values μ_1 and μ_2. We can estimate the true distance by averaging the returned values while giving more confidence to the most accurate sensor, i.e., the one with the smaller σ_i. Hence, in averaging, μ_i is weighted by $1/\sigma_i$. The estimated distance

μ is associated with a standard deviation σ defined in Equation 20.10. This estimates has good properties: it minimizes the mean squared error. The error resulting from the combination of the two measures decreases, since σ <$\min\{\sigma_1, \sigma_2\}$.

$$\begin{aligned} \mu &= \alpha(\mu_1/\sigma_1 + \mu_2/\sigma_2), \text{ with } \alpha = \sigma_1\sigma_2/(\sigma_1 + \sigma_2) \\ 1/\sigma &= 1/\sigma_1 + 1/\sigma_2 \end{aligned} \tag{20.10}$$

SLAM applies the above principle over incremental measures instead of (or in addition to) simultaneous ones. It combines the current estimate (μ', σ') with the new measures (μ_z, σ_z) to compute a new estimate at time t (μ_t, σ_t). This is done with the above equation, rearranged easily into the following incremental form (Equation 20.11):

$$\begin{aligned} \mu_t &= \mu' + K(\mu_z - \mu') \\ \sigma_t &= \sigma' - K\sigma' \\ K &= \sigma'/(\sigma_z + \sigma') \end{aligned} \tag{20.11}$$

Let us now consider the robot's motion. At time $t-1$ the robot was in a position estimated by $(\mu_{t-1}, \sigma_{t-1})$. Between $t-1$ and t the robot moves according to a command known with an uncertainty similarly modeled. Let $(\boldsymbol{u}_t, \sigma_u)$ be this motion estimated from the command and the sensors. The relative distance to the landmark after the motion is estimated by (μ', σ'). The error increases due to the motion:

$$\begin{aligned} \mu' &= \mu_{t-1} + \boldsymbol{u}_t \\ \sigma' &= \sigma_{t-1} + \sigma_u \end{aligned} \tag{20.12}$$

We now can combine the two previous steps. The estimate of the relative position robot - landmark is updated between $t-1$ and t in two steps:

(i) update due to motion (with Equation 20.12) : $(\mu_{t-1}, \sigma_{t-1}) \rightarrow (\mu', \sigma')$
(ii) update due to sensing (with Equation 20.11) : $(\mu', \sigma') \rightarrow (\mu_t, \sigma_t)$

These updates are applied to vectors (in 2D or 3D) of robot and landmark positions. The position of the robot is that of its local frame attached to the base, with respect to which are positioned its sensors and limbs. The map is characterized by many landmarks positioned in space. A vector μ_t, whose components are the robot and landmark positions, is updated at each step. The error is no longer a scalar σ_t but a covariance matrix Σ_t whose element σ_{ij} is the covariance components i and j of the parameters of μ. The robot position error is coupled to the errors in the map and symmetrically. Now, Kalman filtering applies only to linear relations, while the relationship between the command and the motion is not linear. This constraint can be addressed by linearizing around small motions. This leads finally to the *extended Kalman filter* for the two update steps:

$(\mu_{t-1}, \Sigma_{t-1}) \rightarrow (\mu', \Sigma')$, with vector u_t and matrices A and B for the motion,
$(\mu', \Sigma') \rightarrow (\mu_t, \Sigma_t)$ with vector μ_z and matrix C for the new measurements.

The first step uses the motion to update the positions of the robot and those of the landmarks. The second step integrates the new measurements for both, the localization and mapping. This is done with Equation 20.13:

$$\begin{aligned}
\mu' &= A\mu_{t-1} + B\boldsymbol{u}_t \\
\mu_t &= \mu' + K_t(\mu_z - C\mu') \\
K_t &= \Sigma' C^T (C\Sigma' C^T + \Sigma_z)^{-1} \\
\Sigma' &= \sigma_{t-1} + \Sigma_u \\
\Sigma_t &= \Sigma' - K_t C \Sigma'
\end{aligned} \tag{20.13}$$

where Σ_u and Σ_z account for the motion and the measurements covariances.

The approach converges asymptotically to the true map, with a residual error due to initial inaccuracies. It maintains incrementally an estimate of the robot localization as well as of a bound on the localization error. This bound is critical in navigation. If it grows beyond a threshold, specific actions have to be taken such as stop navigating toward the goal and seek known landmarks.

In the mapping part of SLAM, the robot may add new landmarks. It can also maintain a list of candidate landmarks which are not integrated into the map (nor in the vector μ) until a sufficient number of observations of these landmarks have been made. If n is the dimension of the vector μ (*i.e.*, the number of landmarks), the complexity of the update by Equation 20.13 is $O(n^2)$. The computations can be done online and on board of the robot for n in the order of 10^3, which means a quite sparse map.

Particle filtering SLAM. Particle filtering offers an alternative to Kalman filtering with additional advantages. Instead of estimating the Gaussian parameters (μ, Σ), the corresponding probability distributions are estimated through random sampling. Let $P(X_t|z_{1:t}, \boldsymbol{u}_{1:t}) = \mathcal{N}(\mu_t, \Sigma_t)$ a normal distribution, where X_t is the vector of the robot and landmark positions at the time t, $z_{1:t}$ and $\boldsymbol{u}_{1:t}$ are the sequences of landmark measures and movements from 1 to t. Similarly $P(z_t|X_{t-1}) = \mathcal{N}(\mu_z, \Sigma_z)$.

Let us decompose the state vector X_t into two components related to the robot and the landmarks: $X_t = [\boldsymbol{x}_t, \boldsymbol{y}_1, ..., \boldsymbol{y}_n]^\top$, where $\boldsymbol{x}_t$ is the position vector of the robot at time t, and $\boldsymbol{y} = [\boldsymbol{y}_1, ..., \boldsymbol{y}_n]^\top$ the position of vectors landmarks, which do not depend on time because the environment is assumed static. The usual rules of joint probabilities entail the following:

$$\begin{aligned}
P(X_t|z_{1:t}, \boldsymbol{u}_{1:t}) &= P(\boldsymbol{x}_t|z_{1:t}, \boldsymbol{u}_{1:t}) P(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n|z_{1:t}, \boldsymbol{u}_{1:t}, \boldsymbol{x}_t) \\
&= P(\boldsymbol{x}_t|z_{1:t}, \boldsymbol{u}_{1:t}) \prod_{i=1,n} P(\boldsymbol{y}_i|z_{1:t}, \boldsymbol{x}_t)
\end{aligned} \tag{20.14}$$

The second line results from the fact that, given the position $\boldsymbol{x}_t$ of the robot, the positions of the landmarks do not depend on $\boldsymbol{u}$ and are conditionally independent. The robot does not known precisely $\boldsymbol{x}_t$ but it assumes that $\boldsymbol{x}_t \in R_t = \{\boldsymbol{x}_t^{(1)}, \ldots, \boldsymbol{x}_t^{(m)}\}$, a set of m position hypotheses (or particles). Each hypothesis $\boldsymbol{x}_t^{(j)}$ is associated with a weight $w_t^{(j)}$. R_t and the corresponding weights are computed in each transition from $t-1$ to t by the following steps:

- *Propagation*: for m' positions in R_{t-1} randomly sampled according to the weights $w_{t-1}^{(j)}$, compute the position $\boldsymbol{x}_t^{(j)}$ at time t of resulting from the movement $\boldsymbol{u}_t$, with $m' > m$,
- *Weighting* : the weight $w_t^{(j)}$ of particle $\boldsymbol{x}_t^{(j)}$ is computed taking into account the observation z_t from the product $P(z_t|\boldsymbol{y}, \boldsymbol{x}_t^{(j)})P(\boldsymbol{y}|z_{1:t-1}, \boldsymbol{x}_{t-1}^{(j)})$.
- *Sampling*: the m most likely assumptions according to the new weights $w_t^{(j)}$ are kept in R_t.

For each of the m particles, the probability $P(\boldsymbol{y}_i|z_{1:t}, \boldsymbol{x}_t)$ is computed with a Kalman filter reduced to the 2 or 3 parameters necessary to the position $\boldsymbol{y}_i$. With good data structures for the map, this approach, called FastSLAM, reduces the complexity of each update to $O(n\log m)$ instead of $O(n^2)$ in the previous approach. In practice, one can keep a good accuracy for about $m \simeq 10^2$ particles, allowing to maintain online a map with $n \simeq 10^5$ landmarks.

Data association. The main limitation of Kalman and particle filtering approaches is due to a well-known problem of *data association*. At each step of the incremental localization process, one must be sure not to confuse the landmarks: associated measurements should be related to the same landmark. An update of the map and the robot positions with measurements related to distinct landmarks can lead to important errors, well beyond the sensory-motor errors. This argument, together with the computational complexity issue, favors sparse maps with few discriminating and easily recognizable landmarks. On a small motion between $t-1$ and t, the landmarks in the sensory field of the robot are likely to be recognized without association errors. But after a long journey, if the robot views some previously seen landmarks, a robust implementation of the approach requires a good algorithm for solving the data association problem. In the particle filtering approach, the probability distribution of R_t is very different when the robot discovers a new place (equally likely distribution) from the case where it retraces its steps to an already known area.[15] This fact is used by active mapping approaches, which make the robot retrace back its steps as frequently as needed.

In the general case, there is a need for an explicit data association step between the two stages *(i)* and *(ii)* corresponding to Equation 20.13. This step leads to maintain multiple association hypotheses. The SLAM approaches with Dynamic Bayesian Networks (DBN, see Section 8.2.2) for handling multi-hypotheses give good results. The DBN formulation of SLAM results in a dependency graph (Figure 20.11) and the

[15]This is sometimes referred to as the *SLAM loop problem*.

$$\begin{tikzcd}
u_{t-1} \arrow[d] & u_t \arrow[d] & u_{t+1} \arrow[d] \\
r_{t-1} \arrow[r] \arrow[d] & r_t \arrow[r] \arrow[d] & r_{t+1} \arrow[d] \\
z_{t-1} & z_t & z_{t+1} \\
\phi \arrow[u] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

Figure 20.11. Formulation of SLAM with a dynamic Bayesian network; arcs stand for conditional dependencies between random variables, $\boldsymbol{y}$ gives the positions of the landmarks (time-independent), $\boldsymbol{u}_t, \boldsymbol{x}_t$ and z_t denote the movement, the robot positions and the new measurements at time t.

following recursive equation:

$$\begin{aligned} P(X_t|z_{1:t}, \boldsymbol{u}_{1:t}) &= \alpha P(z_t|X_t) \int P(X_t|\boldsymbol{u}_t, X_{t-1}) P(X_{t-1}|z_{1:t-1}, \boldsymbol{u}_{1:t-1}) dX_{t-1} \\ &= \alpha P(z_t|X_t) \int P(\boldsymbol{x}_t|\boldsymbol{u}_t, \boldsymbol{x}_{t-1}) P(X_{t-1}|z_{1:t-1}, \boldsymbol{u}_{1:t-1}) d\boldsymbol{x}_{t-1} \end{aligned} \tag{20.15}$$

Here, α is a simple normalization factor. The vector state is as above $X_t = [\boldsymbol{x}_t, \boldsymbol{y}_1, ..., \boldsymbol{y}_n]^\top$; the second line results from the fact that the environment is assumed static and that the robot motion and landmark positions are independent. The term $P(z_t|X_t)$ expresses the sensory model of the robot, and the term $P(\boldsymbol{x}_t|\boldsymbol{u}_t, \boldsymbol{x}_{t-1})$ corresponds to its motion model. This formulation is solved by classical DBN techniques, using, e.g., the Expectation-Maximization algorithm (EM), which provides a correct solution to the data association problem. However, online incremental version of EM are quite complex.

Recent approaches to SLAM favor this DBN formulation with a global parameter estimation problem overs the set of landmarks and robot positions. The problem is solved by robust optimization methods. This general formulation, called the beam adjustment method, uses techniques of computer vision and photogrammetry. Visual SLAM has also benefited from recent image processing features which can be quite robust for the localization and identification of landmarks.

Let us conclude this section by mentioning a few possible representations for the map of the environment. Landmarks can be any set of sensory attributes that are recognizable and localizable in space. They can be compound attributes of shapes or more complex objects. The most appropriate attributes are generally specific to the type of sensors used. The global map can be represented as a 2D occupancy grid. Simple 3D maps for indoor environments, such as the Indoor Manhattan Representation, combine vertical planes of walls between two horizontal planes for the floor and ceiling. They can be used with more elaborate representations integrating semantic and topological information, discussed next.

20.3.2 Navigation and Exploration with Semantic Maps

Hybrid navigation. Previous approaches are limited to metric maps. They only handle distances and positions in a global Cartesian frame. When the environment is large, it is important to explicitly represent its topology, possibly associated with semantic information. In this case, a map relies on hierarchical hybrid representations, with metric sub-maps in local Cartesian frames, together with relationships and connectivity constraints between sub-maps. The robot re-locates itself metrically when arriving in a sub-map.

Navigation in this case is hybrid. Within a sub-map, motion techniques are used. Between sub-maps other methods can be more relevant, e.g., road following when the environment is equipped with roads, or magnetic and radio heading between waypoints when available. Sensory aspects and place recognition play an important role in navigation methods.

Mapping and map updates can be as flexible as in the case of SLAM through the updates of a graph of local sub-maps. Topological navigation relies on graph search algorithms, associated with motion techniques in sub-maps. Both techniques can be combined incrementally. Topological methods gives a route which is updated and smoothed incrementally to optimize the motion giving the observed terrain while moving.

Topological planning in a graph or within a grid can be used with a partial knowledge of the environment. Graph search algorithms can be extended to compute paths in a graph, while updating the topology and costs parameters from sensing.

Hybrid topological/metric approaches raise the issue of the frontiers between levels and their granularity. Labels of places (doors, rooms, corridors) and topology can emerge from sensing and/or from a uniform description of space into cells (grids, polygons or Delaunay triangles). Recursive decomposition techniques by *quadtrees* are useful but computationally demanding.

Hybrid exploration. In some case a robot has to explore an unknown or partially known environment, to build or update a hybrid hierarchical map, to find an area or an object of interest, or to inspect its environment. Here, no target configuration $\boldsymbol{\theta}_g$ is given. The robot has to find good view points for an efficient exploration. The problem referred to as "*next best viewpoint*" selection has been studied extensively for object modeling and recognition, and more recently addressed for environment exploration.

Simple approaches rely on a heuristic estimate of an information gain utility function. Exploration follows an incremental greedy search using this heuristic. However, it is possible to plan ahead the exploration by optimizing the joint utility of a sequence of observation poses. Both approaches rely on costly sampling and simulation of numerous viewpoints. Generative neural nets open promising improvement perspectives.

20.4 Manipulation

A robot task is seldom limited to the motion and navigation, it also requires manipulating physically objects, which is the topic of this section.

20.4.1 End-Effectors and Grasps

Manipulation is about applying motion and forces to change the state of objects in the environment. Simple manipulation actions are grasping and carrying. Other actions can also be used to manipulate objects, e.g., pushing, rolling, switching, flipping, pivoting, turning, screwing, throwing, etc.

Manipulation requires modeling the end-effectors. Specific grippers (e.g., hooks, magnetic or air sucking devices, encircling chains, surgical effectors) are adapted to specialized manipulations. Universal hands are more difficult to control but cover a large spectrum of tasks. Two-fingered grippers are among the simplest (Figure 20.4). Figure 20.12 shows an early dexterous three-fingered hand, and a more recent five-fingered one. Justin can catch balls thrown from few meters away with its four-fingered hands (Figure 20.6(a)).

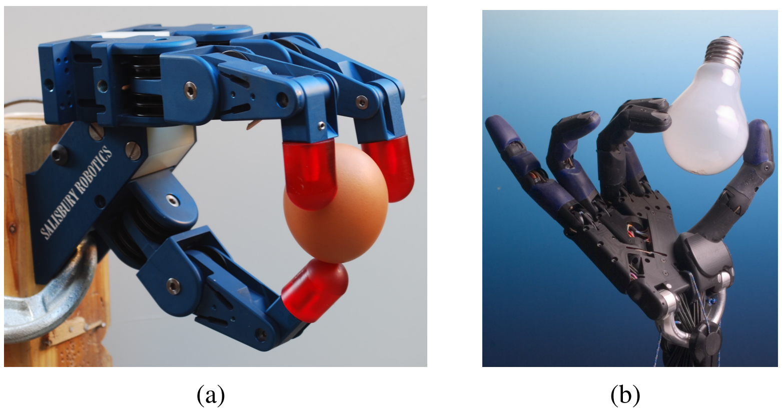


Figure 20.12. (a) The *Salisbury hand*, a pioneering contribution to manipulation with a 9 *dof* three-fingered hand [761]; (b) The *Shadow Robot* five-fingered hand with 20 actuated *dof* out of 24 *dof* [1107].

Computational manipulation is about contact relations between a robot and an object. It requires modeling and handling the following relations:

- contact kinematics: how the motion of objects in contact is constrained,
- contact dynamics: what forces and frictions are transmitted by contact,
- relations between motion and forces, including gravity and inertia, e.g., pushing or carrying a glass without spilling its content.

Most manipulation models rely on a quasi-static assumption, i.e., a slow enough dynamics at contact establishment such as to make all the involved actuation and gravity forces sum up to zero.

Two polyhedra can be in contact through one or several face-to-face, edge-to-face, vertex-to-face, or possibly edge-to-edge relations. Face-to-face contacts are better for grasping, but not necessarily for other manipulation actions. The vertex-to-edge and vertex-to-vertex relations are difficult to control. However, a contact may start with a vertex or an edge, then, through a rotation of one or the two polyhedra, it may get into face-to-face relations. A stable grasp requires generally several contact relations from opposite sides of the grasped object, in order to encircle it with a partial closure. Position sensing (e.g., with vision or laser) is seldom sufficient for manipulation; *tactile or haptic sensors* which measure contact states, slippage and forces are critical.[16]

Beyond grasps, manipulation may require complex motions, in particular for an assembly task. Consider the task of inserting the red peg of Figure 20.4 into a cylindric hole with a narrow clearance. It requires taking a stable grasp at the top end of the peg, then moving it close to destination, seeking contact of the held peg with the hole edge or the surface near it, and moving the peg, while in contact, through a sequence of movements until insertion. A motion constrained by contact is called *compliant*. Passive compliance relies on flexible devices with springs to account for the uncertainty in position and forces. Active compliant motion performs error handling with the identification of the contact state and feedback on contact forces.

20.4.2 Manipulation Space

When considering solely motion in the free space we made two assumptions: (i) objects keep a fixed pose, (ii) the distance δ of $\mathfrak{R}$ to obstacles considers a contact as a collision to be avoided (Equation 20.3). Manipulation requires to reconsider these assumptions.

The distinction between collision and contact is given by the notion of *interior* and *boundary* points. For rigid objects, collision concern interior points, while contact is about boundary points. Let P be a polyhedra in $\mathbb{R}^3$. A point $x \in P$ is an interior point if there exists an open ball centered at x which is completely contained in P. Let $\text{int}(P)$ be the set of interior points of P, and $P\backslash\text{Int}(P)$ the boundary points of P, i.e., points in the closure of P not interior. As a simple example, consider the set $[0, 1)$ in $\mathbb{R}$; its closure is $[0, 1]$, its set if interior points is $(0, 1)$, its boundary points are $\{0, 1\}$. Let us model a solid as identical to its closure: every point in a solid is either interior or boundary.

Recall that $\mathfrak{R}^{\boldsymbol{\theta}}$ and o^q are the set of points in the task space occupied respectively by the robot, when in configuration $\boldsymbol{\theta}$, and an object o, when in a pose $\mathbf{q}$. Let us redefine C_{obs} as the set of configurations $\boldsymbol{\theta}$ in which the robot is in collision with obstacles, $C_{obs} = \{\boldsymbol{\theta} \in C \mid \mathfrak{R}^{\boldsymbol{\theta}} \cap \text{int}(\mathcal{W}) \neq \varnothing\}$, and $C_{free} = C\backslash C_{obs}$. Now, C_{free} allows for contact. A revised Equation 20.3 gives the distance to an object as:

$$\delta(\mathfrak{R}^{\boldsymbol{\theta}}, o^q) = \begin{cases} -\infty & \text{if } \mathfrak{R}^{\boldsymbol{\theta}} \cap \text{int}(o^q) \neq \varnothing \\ 0 & \text{if } \mathfrak{R}^{\boldsymbol{\theta}} \cap [o^q\backslash\text{int}(o^q)] \neq \varnothing \\ min_{\boldsymbol{x}\in\mathfrak{R}^{\boldsymbol{\theta}},\ \boldsymbol{x}'\in o^q} \|\boldsymbol{x} - \boldsymbol{x}'\|^2 & \text{otherwise} \end{cases} \qquad (20.16)$$

[16]Think about handling objects with thick gloves or cold hands, insensitive to tactile feedback.

The function δ indicates either a collision, a contact or gives the minimum distance between $\mathfrak{R}$ and o. It can be computed with an extension of the collision test procedure discussed earlier. Note however that the contact case allows theoretically for no margin, and hence it is computationally more demanding.

Since a contact between $\mathfrak{R}$ and o can happen in many ways, we have to seek the appropriate configurations $\boldsymbol{\theta}$, poses $\mathbf{q}$, and contact relations between $\mathfrak{R}$ and o for the intended manipulation. Let $Q^o \subseteq \mathbb{R}^6$ be the set of possible poses of an object o. We need to rule out poses in which o is in collision with obstacles (defined as we did for C_{obs}), as well as those in which o is unstable. In a *stable* pose, o can rest still without any force applied to it; i.e., o cannot float in the air or be in pose in which it may slide or fall. Stable poses depend on properties of o (shape, friction, mass distribution) as well as of its support. Let Q^o_{sta} be the set of poses of o that are stable and do not collide with obstacles.

The robot end-effector may grasp an object only in a particular set of configurations. For example, the two-finger gripper in Figure 20.4 may not be able to grasp a peg when their two axis are parallel. We can view a grasp as link between two parts in an articulated chain with a rigid joint.[17] Here, o extends the robot's end-effector. The relation between them when respectively in the poses $\mathbf{q}$ and $\mathbf{q}_{eff}(\boldsymbol{\theta})$ is described through a geometric transformation $\mathcal{H}$ between their two local frames (as in Equation 20.2). Obviously, not any such geometric transformation is a feasible grasp: in configuration $\boldsymbol{\theta}$ the end-effector pose $\mathbf{q}_{eff}(\boldsymbol{\theta})$ can be far from the object pose $\mathbf{q}$ or in a position where o is not graspable. But for a given pose $\mathbf{q}$ of o there can be numerous configurations $\boldsymbol{\theta}$ corresponding to feasible grasps.

Let Q^o_{grasp} be the set of pairs $(\boldsymbol{\theta}, \mathbf{q})$ corresponding to feasible grasp positions. The robot in configuration $\boldsymbol{\theta}$ can grasp o in pose $\mathbf{q}$ iff $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$. To each pair $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$ corresponds a uniquely defined transformation $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}} : C \to Q^o$ giving the pose of o when grasped in that grasp. Initially, $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}) = \mathbf{q}$, i.e., the grasp is performed by closing the end-effector without changing $\boldsymbol{\theta}$ nor $\mathbf{q}$. Once grasped, the link between $\mathfrak{R}$ and o becomes rigid: a motion of $\mathfrak{R}$ from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$ moves o from $\mathbf{q}$ to a pose $\mathbf{q}'$ such that $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}')$.

There can be many feasible grasps that are not reachable without collision when o is in a stable pose $\mathbf{q}$. This is the case when all the pairs $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$ are such that $\boldsymbol{\theta} \notin C_{free}$ because of obstacles. This also the case for free configurations $\boldsymbol{\theta}$ that do not permit a grasp, e.g., an object may not be graspable from its wide face; or for poses $\mathbf{q}$ such $\forall\boldsymbol{\theta}, (\boldsymbol{\theta}, \mathbf{q}) \notin Q^o_{grasp}$, e.g., a disk is graspable only in poses where its edge extends out of its support table. Similarly, given a grasp defined by $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$ and a desired target pose $\mathbf{q}'$, there may or may not be a configuration $\boldsymbol{\theta}'$ allowing to ungrasp o in $\mathbf{q}'$: the transformation $\mathcal{H}^{-1}_{\boldsymbol{\theta},\mathbf{q}}(\mathbf{q}')$ is an inverse kinematic problem that may not have a solution.

Having Q^o_{sta} and Q^o_{grasp} we can now define the *manipulation space* of o as:

$$\mathcal{M}^o = \{(\boldsymbol{\theta}, \mathbf{q}) \mid \boldsymbol{\theta} \in C_{free} \ \wedge \ (\mathbf{q} \in Q^o_{sta} \ \vee \ (\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp})\}$$

$\mathcal{M}^o$ is a set of pairs of a free configuration $\boldsymbol{\theta}$ and and object pose $\mathbf{q}$ that are stable, in

[17] We do not consider complex manipulation actions such flipping an object between fingers and allowing a non-actuated *dof* in a grasp (e.g., in a pendulum effect).

a feasible grasp, or both. The latter case corresponds to the grasping and ungrasping moments. Let us analyze the two main manipulation actions:

- *Grasping* an object o in pose $\mathbf{q} \in Q^o_{sta}$: it requires finding $\boldsymbol{\theta} \in C_{free}$ such that $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$. Once grasped, the object's pose is given by $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$. This pose changes with $\boldsymbol{\theta}$ when the robot moves. The robot shape now integrates o in the grasp position. This changes C and hence C_{free}.
- *Ungrasping* o when held in a grasp given by $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$: it requires finding a configuration $\boldsymbol{\theta}'$ in the new C_{free} such that once unsgraped the pose of o is in a stable position, i.e., $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}') \in Q^o_{sta}$.

Grasping and ungrasping start and end at pairs in $\mathcal{M}^o$ meeting $\mathbf{q} \in Q^o_{sta}$ and $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$. The steps for finding a pair $(\boldsymbol{\theta}, \mathbf{q})$ adequate for a grasp or ungrasp entail solving back and forth the direct problem, from $\boldsymbol{\theta}$ to $\mathbf{q}$, and the inverse problem.

The two infinite sets Q^o_{sta} and Q^o_{grasp} play a critical role. They are seldom defined explicitly. Manipulation algorithms rely instead on testing the associated stability and grasping constraints on sampled configurations and poses, or using such tests to define discretized sets of stability and grasping poses.

Manipulation graph. Consider a graph whose vertices are pairs $(\boldsymbol{\theta}, \mathbf{q}) \in \mathcal{M}^o$, that has two types of edges:

- *transit edges*: they represent paths $\boldsymbol{\theta} \in C_{free}$ in which the object o remains still at a pose $\mathbf{q} \in Q^o_{sta}$, the robot does not carry anything and moves to grasp o;
- *carry edges*: they represent paths in which the robot carries o and moves in order to ungrasp o; the pose of o moves along with $\boldsymbol{\theta}$ on the path.

Manipulation is as alternating sequences of transit and carry edges through vertices $(\boldsymbol{\theta}, \mathbf{q})$ in the manipulation space. At the end of a transit path in a vertex $(\boldsymbol{\theta}, \mathbf{q}) \in \mathcal{M}^o$, the robot performs a grasp. This is pursued with a carry path, at the end of which in another vertex of $\mathcal{M}^o$ the robot ungrasps o. A manipulation may require several ⟨transit, grasp, transfer, ungrasp⟩ sequences, as illustrated in the following example.

Example 20.8. Consider the robot in Figure 20.4 which has to insert the red cylinder in the hole in the yellow table. The only initially feasible grasps put the gripper axis orthogonal to the cylinder axis, but end up in collision at the insertion stage. For this task, the robot has to grasp the object, set it on a vertical pose on the table, regrasp it from the top, move it to the hole in the yellow table then perform the insertion. □

A manipulation action changes the configuration space. When the robot grasps an object, its geometry changes. If it holds a long pole, some configurations may no longer be in C_{free}. The grasped object leaves its previous pose; it may no longer be an obstacle to following movements. The scene graph has to be managed dynamically, with updates of the links between nodes at a manipulation action, associated with updates of the configuration space. Manipulation issues are further developed in Section 21.2.

20.5 Discussion and Bibliographic Notes

Robotics is a very large interdisciplinary field which draws from several engineering areas (mechanical, electrical, control, computer, software engineering) and disciplines (physics, chemistry, biology). A broad coverage of robotics can be found in a comprehensive handbook [1014]. Modeling and control issues are developed in several textbooks [685, 792, 743].[18] A collection of papers focuses on problems at the intersection of Robotics and AI [930].

There is a broad literature covering specialized robotics areas, e.g., manufacturing robots [459]; exploration robots [351] in mining [257], ocean [47, 776] and space [1211]; service robots [455, 422]; driverless vehicles [1097, 1217]; personal robots [915]; medical robots [1081]; or human-carried robots [594].

This chapter discusses several technical issues for which additional references are in order. Computational geometry methods are extensively discussed in [974], with additional concerns for robotics in, e.g., [60]. Scene graphs for environment description linked to sensing are surveyed in [215]. The Unified Robot Description Format (URDF) synthesis and use is the topic of many tutorials and surveys, e.g., [1099]. Kinematics and control issues are extensively covered in the already mentioned handbook and textbooks. Closed-loop control on exteroceptive sensors, as in visual servoing [220], are powerful extensions of the control methods in Section 20.2.4. The potential field methods for movement and obstacle avoidance are studied in [602, 83]. A diversity of biped movements, including acrobatics, are illustrated in e.g., [649, 229].

Mobile robots led naturally to navigation issues. The SLAM problem has been extensively studied and implemented.[19] Kalman Filtering methods attracted more attention [1096, 805, 1051, 333] than particle based methods with DBN and the Expectation-Maximization algorithm [403, 1105]. Visual SLAM has benefited from progress in robust image processing for the localization and identification of landmarks [841, 845]. Navigation has also benefited from a broad set of map description methods, e.g., for hierarchical topological maps [638], semantic maps [650], and hybrid maps [651].

Environment exploration issues are surveyed in [731]. Approaches to the next best viewpoint selection problem for exploration are illustrated in, e.g., [1067] for a greedy method, and [390] for a planning method. Generative neural nets for handling semantics in exploration problems are attracting a growing interest [742, 927, 1063].

Robotics manipulation is a very broad area with significant contributions from [759, 760]. The illustrated 3-fingered hand is due to [761]. Manipulation graphs will be further discussed in the next chapter.

20.6 Exercises

20.1. Prove the results given in Example 20.1 for computing (x, y) from (x', y')

[18]The latter has an online course: https://modernrobotics.northwestern.edu
[19]See, e.g., the software repository: http://www.openslam.org/

20.2. A free solid in space has 6 *dof*. How many *dof* have

- a laser pointer?
- a tray of bottles to be kept vertical?

20.3. Based on Example 20.4, develop the equations for the transition function $\dot{\boldsymbol{\theta}} = \gamma(\boldsymbol{\theta}, u)$ of a unicycle with $\boldsymbol{\theta} = (x, y, \alpha)$ and two actuation variables, u_1 the pedaling rate and u_2 the angular velocity.

20.4. Compute $J^{-1}(\boldsymbol{\theta})$ for the robot in Example 20.5 and its determinant. What are the singularities of this robot in the $(\theta_1, \theta_2, \theta_3)$?

20.5. What is the third order polynomial scaling function ζ of Exercise 20.5.

20.6. Identify the joint types and axis of the 9 *dof* of the three-fingered hand in Figure 20.12(a). Same question for the 24 *dof* of the hand in Figure 20.12(b).

21 Task and Motion Planning

Planning in robotics usually refers to motion and manipulation. In AI it refers to abstract actions. The two problems require distinct mathematical representations. The latter relies on the abstract causal relations from preconditions to effects. The former is concerned with computational geometry, kinematics and dynamics. Clearly, no single approach is sufficient since motion and manipulation are needed for most actions, but planning movements in space cannot handle causality and task organisation. In simple cases one may decouple the two problems: abstract planning produces abstract plans whose actions can be refined, when needed, with motion/manipulation planning. However, the two problems are tightly coupled for complex tasked and crammed environments. For example, furniture movers have to carefully organize their task such as not to block the moving of large pieces.

This chapter is about the integration of planning for motion/manipulation with planning for abstract actions. In this context, we refer to the latter as task planning and to the integration of both as *combined task and motion planning problems* (TAMP). The reader is already familiar with task planning techniques. Section 21.1 introduces the main algorithms for motion planning. Manipulation planning is subsequently introduced . Section 21.3 presents a few approaches specific to TAMP.

21.1 Motion Planning

A robot's movements take place computationally in the configuration space, which is a connected manifold (see Chapter 20). Since $C_{free} = C \setminus C_{obs}$, when there are no obstacles $C_{free} = C$. A movement between two configurations $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g \in C_{free}$ is easily computed as a straight line in the configuration space (Equation 20.4). Motion in environments with sparse obstacles can rely on reactive potential field techniques (Section 20.2.5), with the risk of getting trapped in local minima. Moving with a planned path is safer.

A motion planner will be given *(i)* a domain described with the set $\mathcal{W}$ of fixed and movable objects, *(ii)* a robot $\mathfrak{R}$ specified with its kinematic and dynamic capabilities and constraints, and *(iii)* an initial configuration $\boldsymbol{\theta}_0$ and a goal configuration $\boldsymbol{\theta}_g$. Note we do not need to provide and explicit of set of actions. These are limited to motion, described with *metric operational models* of the possible movements of the robot.

In the simplest case, a motion planner produces a path from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$.[1] Sometime we need a trajectory, i.e., a path labeled with time. A *kinodynamic motion planner* seeks a movement which meets the velocity, acceleration, force and torque constraints of

[1]In that case motion planning and path planning are synonymous.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

the robot, as well as the kinematic and obstacle avoidance constraints. We focus here on a simplified kinematic planning problem where dynamic constraints are ignored.

A robot task is expressed in the task space, not in the configuration space, e.g.,"reach for this object", instead of "put yourself in that configuration". We need to map the goal pose of the end effector $\mathbf{q}_g = [x, y, z, \alpha, \beta, \varphi]$ from the task space to a configuration $\boldsymbol{\theta}_g$ such that $\mathbf{q}_{\boldsymbol{\theta}_g}(\mathit{eff}) = \mathbf{q}_g$. This is the inverse of the problem seen earlier (Section 20.2.2). Let us assume that the kinematic of the $\mathfrak{R}$ allows finding a $\boldsymbol{\theta}_g$ meeting the task goal (which is the case when $\mathfrak{R}$ is kinematically redundant and C_{free} is a connected manifold).

Motion planning problems. A motion planning problem is a tuple $(\mathcal{W}, \mathfrak{R}, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g)$ where $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$ are in C_{free}, the free configuration space of $\mathfrak{R}$. A solution to the problem is a path $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$ in C_{free} from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$.

The configuration space representation makes it possible to conceptually simplify the motion of an articulated body in a complex environment as a simple continuous path of points in the free configuration space. Despite this conceptual simplification, motion planning is a PSPACE-hard problem. A major complexity issue is that the search space C_{free} is not easily computed from the problem specifications.[2] Mapping $\mathcal{W}$ into C to obtain C_{obs}, then C_{free} is intractable even for very simple robots and environments (Section 20.2.2). This issue is practically addressed by defining an explicit discrete roadmap in C_{free} which simplifies motion planning, as developed next.

21.1.1 Planning with a Roadmap of the Free Configuration Space

Since the search space C_{free} is very hard to compute explicitly, possible approaches have to seek approximations, e.g., by discretization (see Section 21.1.6) or Monte Carlo sampling. A fruitful idea relies on the latter to sample a finite set of configurations in C_{free} connected into a graph, such that a graph path can easily be mapped into a path in C_{free} (as defined in Definition 20.3).

Let $\mathcal{G} = (V, E)$ be an undirected graph; its vertices are to points in C_{free}; its edges are *segments* in C_{free} (i.e., not segments in $\mathbb{R}^3$, but simple paths as per Equation 20.4):

- $V = \{\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_n | \boldsymbol{\theta}_i \in C_{\mathit{free}}\}$, and
- for every edge $(\boldsymbol{\theta}_i, \boldsymbol{\theta}_j) \in E$ there is a segment $\wp_{[\boldsymbol{\theta}_i, \boldsymbol{\theta}_j]}$ in C_{free} from $\boldsymbol{\theta}_i$ to $\boldsymbol{\theta}_j$.

Let $\sigma = \langle \boldsymbol{\theta}_{i,1}, \ldots, \boldsymbol{\theta}_{i,k} \rangle$ be a sequence of k distinct vertices in V such that consecutive vertices are edges in E, i.e., $(\boldsymbol{\theta}_{i,j}, \boldsymbol{\theta}_{i,j+1}) \in E$ for $1 \leq j \leq k-1$. This is a graph path in $\mathcal{G}$ which corresponds to a sequence of segment paths: $\langle \wp_{[\boldsymbol{\theta}_{i,1}, \boldsymbol{\theta}_{i,2}]}, \ldots, \wp_{[\boldsymbol{\theta}_{ink-1}, \boldsymbol{\theta}_{ink}]} \rangle$ in C_{free}. Hence σ also corresponds to a path $\wp_{[\boldsymbol{\theta}_{i,1}, \boldsymbol{\theta}_{i,k}]}$ in C_{free}.[3] Note that a path $\wp$ in C_{free} can be decomposed by linear interpolation into a sequence of segments that, in general, may not map to a graph path in $\mathcal{G}$.

[2]The search space is seldom explicit in task as well as in motion planning, but in task planning it can be partially enumerated from s_0 and $\gamma(s, a)$.

[3]This path is a continuous piecewise linear function, its first order derivative is discontinuous at the intermediate points $\boldsymbol{\theta}_{i,1}, \ldots, \boldsymbol{\theta}_{i,k}$.

Definition 21.1. A *roadmap* for C_{free} is a graph $\mathcal{G} = (V, E)$ of vertices and segment edges in C_{free} which meets the two following properties:

(i) *Connectivity*: two vertices are connected in $\mathcal{G}$ iff the corresponding configurations are connected in C_{free}, i.e., $\forall \boldsymbol{\theta}_i, \boldsymbol{\theta}_j \in V$, there is a path σ in $\mathcal{G}$ from $\boldsymbol{\theta}_i$ to $\boldsymbol{\theta}_j$ iff there is a path $\wp_{[\boldsymbol{\theta}_i, \boldsymbol{\theta}_j]}$ in C_{free}, and

(ii) *Accessibility*: every point in C_{free} can be connected to $\mathcal{G}$ in C_{free}, i.e., $\forall \boldsymbol{\theta} \in C_{free}$, there is a vertex $\boldsymbol{\theta}_i \in V$ such that the segment $\wp_{[\boldsymbol{\theta}, \boldsymbol{\theta}_i]}$ is in C_{free}. □

The connectivity property states that the graph connectivity matches the free configuration space connectivity: a path σ in the graph $\mathcal{G}$ corresponds to a continuous path $\wp$ in C_{free}, and vice versa for the vertices of V. Hence connected components of the graph correspond to connected components of C_{free}. The accessibility property states that every configuration in C_{free} is accessible with a segment from $\mathcal{G}$.

Example 21.2. Consider the simple 2D configuration space of Figure 21.1(a): there is path between every pair of points in C_{free}; hence the corresponding roadmap is a connected graph. In Figure 20.8, the free configuration space is not connected, its roadmap will have two connected components □

Given a roadmap $\mathcal{G} = (V, E))$, a motion planning problem $(\mathcal{W}, \mathfrak{R}, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g)$ is easily solved with the Roadmap-MP procedure Algorithm 21.1.

```
Roadmap-MP($\boldsymbol{\theta}_0, \boldsymbol{\theta}_g, V, E$)
    find a vertex $\boldsymbol{\theta}_i \in V$ with a segment $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_i]}$ in $C_{free}$
    find a vertex $\boldsymbol{\theta}_j \in V$ with a segment $\wp_{[\boldsymbol{\theta}_g, \boldsymbol{\theta}_j]}$ in $C_{free}$
    search for a graph path $\sigma$ from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$ in the graph
      $(V \cup \{\boldsymbol{\theta}_0, \boldsymbol{\theta}_g\}, E \cup \{(\boldsymbol{\theta}_0, \boldsymbol{\theta}_i), (\boldsymbol{\theta}_g, \boldsymbol{\theta}_j)\})$
    return $\sigma$ or nil if no path is found
```

Algorithm 21.1. Roadmap-MP, a motion planning with a roadmap

The first two steps of Roadmap-MP connect $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$ to $\mathcal{G}$; they are granted by the accessibility property. If the third step does not find path σ, then the problem has no solution, thanks to the connectivity property. Otherwise the returned path σ in the extended roadmap is mapped to a sequence of segments $\langle \wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_i]}, \ldots, \wp_{[\boldsymbol{\theta}_j, \boldsymbol{\theta}_g]} \rangle$ from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$ in C_{free} that provide the motion plan solution $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$.

In summary, with a roadmap of C_{free}, motion planning problems are solved with a simple path search algorithm in an explicitly defined graph. We transform a PSPACE-hard problem into a simple polynomial problem (graph search) at a cost: we trade complexity for approximate completeness. In practice, Roadmap-MP is only probabilistically complete: it uses *probabilistic roadmaps*, i.e., graphs that approximate a roadmap in a probabilistic sense. As detailed next, the more we sample points in C_{free}, the larger the probability of completeness converges to 1.

21.1.2 Sparse Probabilistic Roadmaps

Given a motion planning problem $(\mathcal{W}, \mathfrak{R}, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g)$, let see how to synthesize a graph $\mathcal{G}$ that probabilistically approximates a roadmap.

All probabilistic roadmap algorithms rely on the same schema: they synthesize a graph by randomly sampling configuration points in C, testing them for the needed properties, and extending $\mathcal{G}$ with vertices and/or edges when the tests are satisfied.

The *visibility-based probabilistic roadmap* algorithm, vPRM (Algorithm 21.2), applies the above schema in a parsimonious way. The schema, illustrated in Figure 21.1, adds to $\mathcal{G}$ a sampled $\boldsymbol{\theta}$ when $\boldsymbol{\theta} \in C_{free}$ and either:

- $\boldsymbol{\theta}$ extends the accessibility to reach parts of C_{free} not yet covered, or
- $\boldsymbol{\theta}$ extends the connectivity to connect unconnected components of $\mathcal{G}$.

A point $\boldsymbol{\theta}$ that meets the former condition extends the coverage with a free *visibility area*, in which any point will "see" $\boldsymbol{\theta}$ with a segment. A point in the latter condition allows connecting two unconnected such areas.

```
vPRM($V, E, n, \eta$)
    $n' \leftarrow n$
    while $n' > 0$ do
        $\boldsymbol{\theta} \leftarrow$ Sample($C$)
        if Free($\boldsymbol{\theta}, \eta$) then
1           if $\forall \boldsymbol{\theta}' \in V$: $\neg$FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}', \eta$) then
                $V \leftarrow V \cup \{\boldsymbol{\theta}\}$
                $n' \leftarrow n$
2           else if $\exists \boldsymbol{\theta}_1, \boldsymbol{\theta}_2$ in $V$ such that $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are not connected in $E$, and
                FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}_1, \eta$) and FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}_2, \eta$) then
                $V \leftarrow V \cup \{\boldsymbol{\theta}\}$
                $E \leftarrow E \cup \{(\boldsymbol{\theta}, \boldsymbol{\theta}_1), (\boldsymbol{\theta}, \boldsymbol{\theta}_2\}$
                $n' \leftarrow n$
            else $n' \leftarrow n' - 1$
    return $\mathcal{G} = (V, E)$
```

Algorithm 21.2. vPRM, a visibility-based probabilistic roadmap algorithm for path planning

The algorithm vPRM is initially called with $V = \varnothing$, $E = \varnothing$, a collision margin threshold η, and a positive integer n for controlling its termination. vPRM uses the following functions:

- Sample(C): returns a uniformly distributed random point $\boldsymbol{\theta} \in C$. Recall that a kinematic model gives $C \subseteq \mathbb{R}^n$ for a robot with n degrees of freedom (*dof*). Several calls to Sample return a set of points that are independently and identically distributed (the *i.i.d* assumption).
- Free($\boldsymbol{\theta}, \eta$): returns "true" if $\boldsymbol{\theta} \in C_{free}$ up to the error margin threshold η, and "false" otherwise.

- FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}', \eta$): returns "true" if the segment path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ is in C_{free} up to the error margin η, and "false" otherwise. Recall that η is the error margin allowed in computing the distance to obstacles δ (p.454).

The collision testing procedure Free($\boldsymbol{\theta}, \eta$) relies on the distance δ to obstacles (Equation 20.3). It can be efficiently implemented with hierarchical geometric data structures, e.g., trees of bounding boxes (see Section 20.2.2). Recall the error margin parameter η: the farther the robot can be from obstacles, the more efficient collision testing is. Conversely, the lower is the error margin threshold η, the higher is the cost of the collision test. Free can be extended to a procedure FreeSegment for testing if a segment is in C_{free}: since a segment path is simple linear interpolation between two poins (Equation 20.4), this test can be performed by sampling on the segments taking into account the distance δ to obstacles.

The graph $\mathcal{G}$ can be maintained such as to ease the tests in lines 1 and 2, e.g., order the vertices of V and keep track of the connected components of $\mathcal{G}$ to test with a Union-Find algorithm if $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are connected.

The termination condition is based on the number n of *consecutive* samples of free configurations that do not add anything to the roadmap, i.e., samples that fail the tests in lines 1 and 2. vPRM is probabilistically complete; probability that the resulting graph is a roadmap of C_{free} is proportional to $(1 - 1/n)$.

Example 21.3. Let us use vPRM to build a roadmap of a 2D configuration space with the grey obstacles shown in Figure 21.1(a). Figure 21.1(b) to (h) illustrate a few processing stages, showing only the points in C_{free}, ignoring samples in C_{obs}:

(b): point 1 is added to V, but not point 2, nor any other point in the visibility area (in blue) covered by point 1, unless it extends the graph connectivity.
(c): point 3 is added to V covering the green visibility area, but points 4 and 5 are ignored.
(d): point 6 is added to V covering the brown visibility area, but not points 7, 8 and 9.
(e): point 10, although in an area already covered by points 3 and 6, is added to V because it extends the connectivity; two edges are added to E; points 11 and 12 are ignored.
(f): point 13 is added to V (yellow area), but not point14.
(g): point 15 and two edges are added to $\mathcal{G}$.
(h): point 17 and two edges are added to $\mathcal{G}$.

At this stage no further samples can extend the graph coverage. The algorithm stops after n consecutive unsuccessful trials, with a roadmap of 7 vertices and 6 edges (Figure 21.1(h)). □

In vPRM, a roadmap is a sparse graph to which a new vertex is added whenever the tests in lines 1 and 2 show that the vertex will extend the roadmap's coverage. One may even go further in seeking a minimal roadmap by exchanging a vertex already in V with a newly sampled but possibly better $\boldsymbol{\theta} \in C_{free}$. No efficient heuristic for doing so has been devised, and it is unclear whether the additional computation would be amortized over numerous uses of the roadmap.

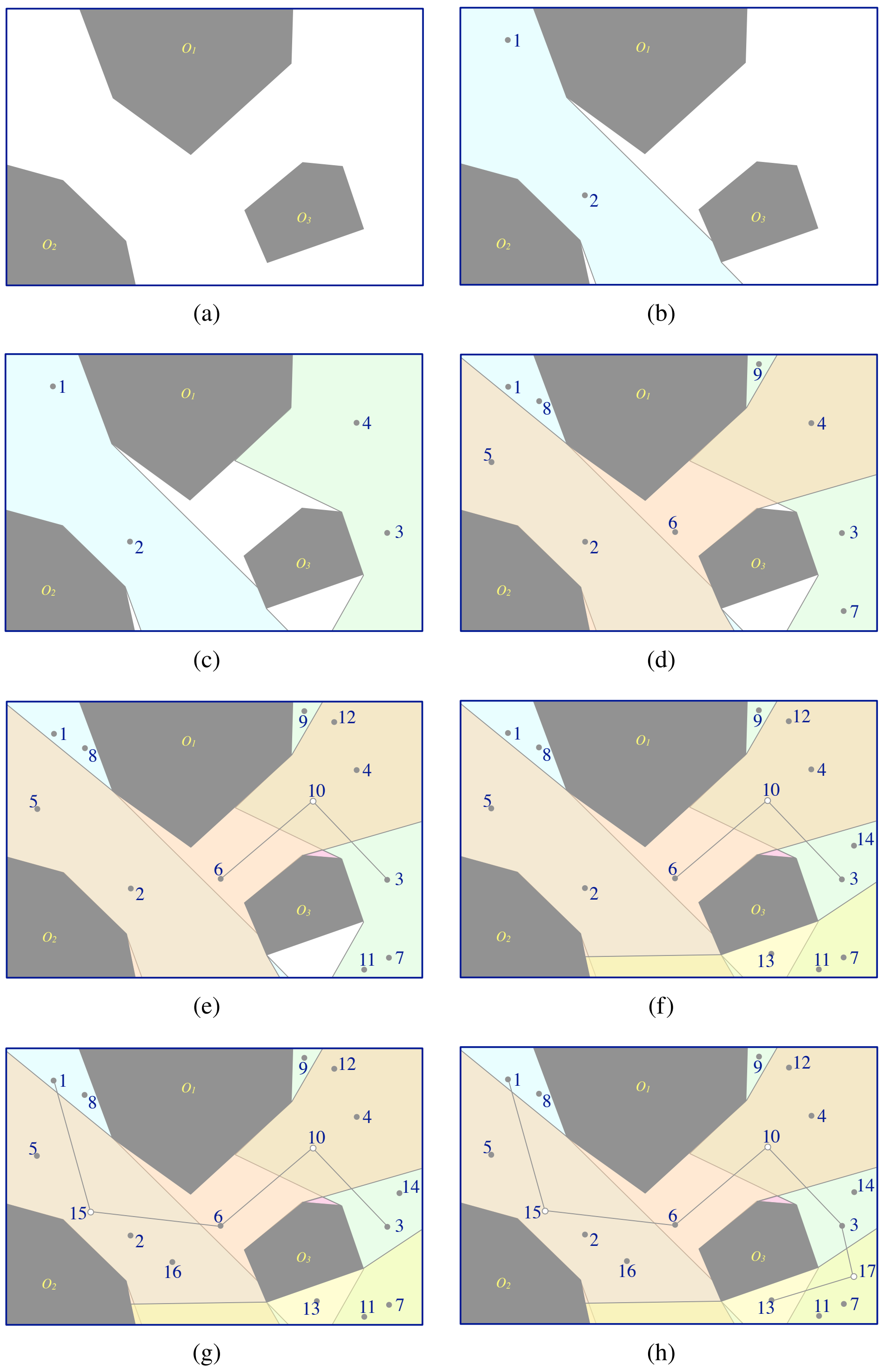


Figure 21.1. Illustration of the vPRM algorithm in a 2D configuration space.

21.1.3 Redundant Probabilistic Roadmaps

One may prefer to synthesize quickly a very large redundant roadmap. This motivation drives the sPRM algorithm (Algorithm 21.3). sPRM adds to V every sampled $\boldsymbol{\theta} \in C_{free}$ until reaching n vertices, then for each vertex $\boldsymbol{\theta} \in V$ it adds to E all free segments that connect $\boldsymbol{\theta}$ to its neighbors, within a neighborhood of radius ϵ from $\boldsymbol{\theta}$ (line 1).

```
sPRM($V, E, n, \eta, \epsilon$)
    while $|V| \leq n$ do
        $\boldsymbol{\theta} \leftarrow$ Sample($C$)
        if Free($\boldsymbol{\theta}, \eta$) then $V \leftarrow V \cup \{\boldsymbol{\theta}\}$
    foreach $\boldsymbol{\theta} \in V$ do
1       foreach point in $\{\boldsymbol{\theta}' \in V \mid \|\boldsymbol{\theta} - \boldsymbol{\theta}'\|^2 \leq \epsilon\}$ do
            if FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}', \eta$) then $E \leftarrow E \cup \{(\boldsymbol{\theta}, \boldsymbol{\theta}')\}$
    return $\mathcal{G} = (V, E)$
```

Algorithm 21.3. sPRM, a simplified probabilistic roadmap algorithm

sPRM, like vPRM, grants probabilistic completeness. The probability that the returned $\mathcal{G}$ is a roadmap of C_{free} equal to $(1 - e^{-cn})$, for some constant c. Intuition and detailed analysis show that the neighborhood parameter ϵ is better set as a function of n decreasing with a rate in $O(\log(n))$.

Both vPRM and sPRM synthesize a roadmap as an investment to be used over multiple motion planning calls of Roadmap-MP. In applications where the environment is changing, e.g., due to the robot's own actions, this investment for finding an almost complete roadmap of C_{free} is not worthwhile. One may restrict the search to a reduced graph, focused on the specific motion planning problem from $\boldsymbol{\theta}_0$ to $\boldsymbol{\theta}_g$ (see Section 21.1.5).

21.1.4 Incremental Roadmap Refinement

The previous algorithms face the issues of probabilistic completeness and the complexity of finding paths close to obstacles. Recall that the closer to obstacles the robot needs to move, the more the collision testing will cost. This is illustrated in the next example. As for any incomplete algorithm, the failure of Roadmap-MP is inconclusive: there may be a solution, but the generated $\mathcal{G}$ might not allow it to be found. One may further extend the roadmap, that is, pursue the procedure with a larger parameter n that controls the termination condition. Moreover, the error margin parameter η for collision testing plays also a critical role. A cautious large margin may forbid possible roads. The smaller η is, the closer to obstacles sampled points can be in C_{free} and contribute to $\mathcal{G}$.

Example 21.4. Consider Figure 21.1 with a much larger upper obstacle o_1, leaving only narrow free corridors to the two other obstacles. If the corridor is narrower than the proximity margin of the collision test, points such as 2, 6, and 15 would not be considered as in C_{free}. Hence vPRM or sPRM, even with a large n, would give a roadmap that leaves connected parts of C_{free} unconnected. They would not be able to tell whether there is no connection or there is one that has not been found. □

```
Incremental-MP($\theta_0, \theta_g, n, \eta, \Delta n, \rho_\eta$)
    $(V, E) \leftarrow$ PRM($\varnothing, \varnothing, n$)
    $\sigma \leftarrow$ Roadmap-MP($\theta_0, \theta, V, E$)
    repeat
        $n \leftarrow n + \Delta n$ ; $\eta \leftarrow \eta/\rho_\eta$
        $(V, E) \leftarrow$ PRM($V, E, n, \eta$)
        $\sigma \leftarrow$ Roadmap-MP($\theta_0, \theta, V, E$)
    until $\sigma \neq$ nil or timeover
```

Algorithm 21.4. Incremental-MP, motion planning with incremental refinements of the roadmap, with *PRM* being either vPRM or sPRM.

To address this issue, vPRM or sPRM can be used with incremental refinements of the roadmap, starting with a rough map and augmenting its resolution until either a solution is found or planning time is over. This is done in Algorithm 21.4. Initially the error parameter η is set to a comfortable margin to reduce the computational cost of the collision testing in C_{free}. In case of failure, the roadmap is extended with a larger n and a smaller η.

Probabilistic roadmap algorithms are quite efficient for sparsely occupied environments, but much less for very cluttered ones.

21.1.5 Rapidly-Exploring Random Trees

The PRM methods in the previous section invest in the construction of a roadmap of C_{free}. Its computation cost is amortized over many motion-planning problem instances in the same environment. We now discuss faster and more focused algorithms that seek a solution path for a single motion-planning instance of moving between two points θ_0 and θ_g.

A Rapidly-Exploring Random Tree algorithm (RRT) builds a tree rooted at θ_0 with a leaf at θ_g. It is called a *single query* motion planner, as opposed to the multiple queries with the same $\mathcal{G}$ that motivate PRM algorithms. The synthesized tree is to be used just once.

The algorithm RRT is initially called with $V = \{\theta_0\}$ and $E = \varnothing$. In each iteration of its loop, it randomly selects a point $\theta \in C_{free}$, then tries to connect θ to the nearest point $\theta' \in V$. The steps are as follows:

- RRT first calls Select, which returns the goal θ_g with probability p, and otherwise returns a new sampled point in C_{free}.
- Nearest(θ, V) returns θ' the nearest point to θ in the set of configurations V, that is, Nearest$(\theta, V) = \text{argmin}_{\theta' \in V} \|\theta - \theta'\|^2$.
- If the segment $\wp_{[\theta,\theta']}$ is in C_{free}, that is, FreeSegment(θ, θ', η) is true, the path $\wp_{[\theta,\theta']}$ connects θ to V, so RRT sets $\theta_{new} \leftarrow \theta$.

```
RRT($V, E, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g, n, \eta, p$)
    until ($\boldsymbol{\theta}_g \in V$) or $|V| = n$ do
        $\boldsymbol{\theta} \leftarrow$ Select($\boldsymbol{\theta}_g, p$)
        $\boldsymbol{\theta}' \leftarrow$ Nearest($\boldsymbol{\theta}, V$)
        if FreeSegment($\boldsymbol{\theta}, \boldsymbol{\theta}', \eta$) then $\boldsymbol{\theta}_{new} \leftarrow \boldsymbol{\theta}$
        else $\boldsymbol{\theta}_{new} \leftarrow$ Steer($\boldsymbol{\theta}, \boldsymbol{\theta}'$)
        $V \leftarrow V \cup \{\boldsymbol{\theta}_{new}\}$; $E \leftarrow E \cup \{(\boldsymbol{\theta}', \boldsymbol{\theta}_{new})\}$
    return $\mathcal{G} = (V, E)$
```

Algorithm 21.5. RRT, Rapidly-Exploring Random Tree algorithm

- Otherwise, RRT uses Steer($\boldsymbol{\theta}, \boldsymbol{\theta}'$) to return a point $\boldsymbol{\theta}_{new}$ between $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ for which there is a segment in $\mathcal{C}_{free}$. The Steer function chooses this point randomly according to some heuristic, e.g., in between the nearest obstacle to $\boldsymbol{\theta}'$ in the direction of $\boldsymbol{\theta}$ and midway to $\boldsymbol{\theta}'$ (Figure 21.2).
- In either case, RRT modifies V and E to add the connection to $\boldsymbol{\theta}_{new}$.

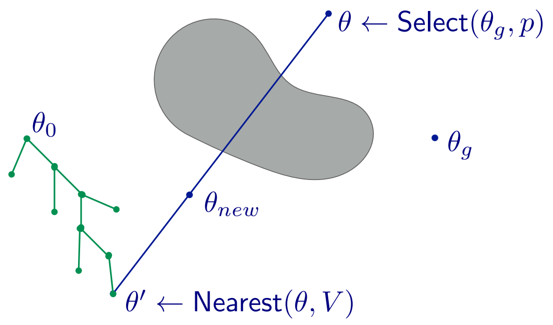

$\theta \leftarrow$ Select(θ_g, p)
θ_0
θ_g
θ_{new}
$\theta' \leftarrow$ Nearest(θ, V)

Figure 21.2. Tree built by RRT algorithm (in green); added vertex and edge $\boldsymbol{\theta}_{new}$ and $(\boldsymbol{\theta}', q_{new})$

```
Select($\boldsymbol{\theta}_g, p$)
    switch randomly do
        case with probability $p$ do return $\boldsymbol{\theta}_g$
        case with probability $1 - p$ do
            repeat $\boldsymbol{\theta} \leftarrow$ Sample($\mathcal{C}$) until $\boldsymbol{\theta} \in \mathcal{C}_{free}$
            return $\boldsymbol{\theta}$
```

Algorithm 21.6. Selects randomly either $\boldsymbol{\theta}_g$ with a probability p, or a sampled free configuration with a probability $1 - p$.

The Select function guides flexibly the synthesis of the tree towards the goal. The probability parameter p can be set as increasing from a low value (e.g., 10^{-2}) at the beginning of the search to allow for a random growth of the tree, to a higher value later on to close up on the goal. When p is high enough, the termination condition simply waits until $\boldsymbol{\theta}_g$ is added to $\mathcal{G}$.

The graph constructed by RRT is a tree, because it adds new edges only for new vertices that are not already in V. This tree is rooted at $\boldsymbol{\theta}_0$. If RRT terminates with $\boldsymbol{\theta}_g \in V$ then there is a single path $\sigma = \langle \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_g \rangle$ in $\mathcal{G}$ to the goal configuration.

The considerations of Section 21.1.4 about handling probabilistic completeness and proximity to obstacles also apply to RRT. The algorithm Incremental-RRT is an incremental version which repeatedly calls RRT from the previously returned V and E with larger n and p and a smaller η until either a solution is found or planning time is over.

```
Incremental-RRT($\boldsymbol{\theta}_0, \boldsymbol{\theta}_g, n, \eta, p, \Delta n, \rho_\eta, \Delta p$)
    $(V, E) \leftarrow$ RRT($\{\boldsymbol{\theta}_0\}, \varnothing, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g, n, \eta, p$)
    repeat
        $n \leftarrow n + \Delta n$ ; $\eta \leftarrow \eta/\rho_\eta$ ; $p \leftarrow p + \Delta p$
        $(V, E) \leftarrow$ RRT($V, E, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g, n, \eta, p$)
    until $\boldsymbol{\theta}_g \in V$ or timeover
    if $\boldsymbol{\theta}_g \in V$ then return the path $\sigma = \langle \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_g \rangle$ in $\mathcal{G}$
    else return nil
```

Algorithm 21.7. Incremental-RRT, motion planning with an RRT algorithm

There are several variants of the RRT algorithm. For example, the Rapidly-Exploring Dense Tree (RDT) algorithm simply keeps adding to V sampled free points and connecting them to their nearest neighbor in V, without aiming for $\boldsymbol{\theta}_g$. A more elaborate variant is the bi-directional RRT, which grows concurrently two trees rooted at $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$, until the two can be connected. All these algorithms are probabilistically complete.

Transforming a graph path into a smooth trajectory. PRM and RRT methods provide a plan as a path $\sigma = \langle \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_g \rangle$ in a graph whose vertices and edges are in $\mathcal{C}_{free}$. However, the motion of the robot requires a path $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$ that is a continuous function from $[0, 1]$ to $\mathcal{C}_{free}$ (see Definition 20.3). This path $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$ will be further transformed into a trajectory with a time-scaling function (Section 20.2.4).

The mapping from σ to $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$ is a classical problem of fitting a curve to a set of points in $\mathbb{R}^n$, using, e.g., nonlinear regression techniques with polynomials or other smooth functions. We further need to check that the resulting curve remains in $\mathcal{C}_{free}$.

Collision checking is generally easier for a segment than for a curve. To simplify checking $\wp_{[\boldsymbol{\theta}_0, \boldsymbol{\theta}_g]}$, and prior to curve fitting, it can be worthwhile to interpolate σ with additional points and segments in $\mathcal{C}_{free}$. Let $\boldsymbol{\theta}_{i-1}, \boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1}$ be three consecutive vertices in σ, $\boldsymbol{\theta}'_{i-1}$ and $\boldsymbol{\theta}'_i$ two points on the segments $[\boldsymbol{\theta}_{i-1}, \boldsymbol{\theta}_i]$ and $[\boldsymbol{\theta}_i, \boldsymbol{\theta}_{i+1}]$ respectively. If $\boldsymbol{\theta}'_{i-1}, \boldsymbol{\theta}'_i$ and the segment $[\boldsymbol{\theta}'_{i-1}, \boldsymbol{\theta}'_i]$ are in $\mathcal{C}_{free}$, then $\boldsymbol{\theta}_i$ can be replaced in σ by the pair $\boldsymbol{\theta}'_{i-1}, \boldsymbol{\theta}'_i$, with the associated three segments instead of two. This procedure can be repeated recursively so as to have only short segments in σ. If the distance of the fitted curve to the segments in σ is below the error margin η then no further collision checking is needed (see Exercise 21.2).

Coupling RRT to PRM and to potential fields. Recall that potential field methods use a repulsive potential ϕ to define a reactive policy $\pi(\boldsymbol{\theta}) = -\nabla\phi(\boldsymbol{\theta})$ (see Section 20.2.5). Their advantage is the coupling to online sensing to avoid non-modeled and moving obstacles, but they are incomplete and may get trapped in local minima. A natural idea is to plan the motion with RRT, for example, and use the graph path σ as a sequence of intermediate goals in C_{free} to a potential field method. Planning may take into account only fixed and large obstacles, and possibly be further refined if the potential field fails locally.

21.1.6 Planning in a Discretized Configuration Space

Assume that the configuration space is discretized into small adjacent cells that all fit into C_{free}. With such a discretization, motion planning is reduced to a simple search of a path in the adjacency graph, which can be solved with classical graph search algorithms. The issue here is to efficiently find a good discretization.

A simple and popular grid-based method can be used as a first approximation for a mobile robot. For example, a mobile robot would plan a rough motion of its base in a grid; when close to obstacles and goals, it would refine the plan with more elaborate algorithms for the motion of its arms and limbs. Starting with a uniform partition of space into a grid, the method labels as free the cells that do not intersect with obstacles. Collision checking is needed for a cell's corners and edges. Note that even with convex polygonal obstacles, it is not sufficient to have the four corners of a cell in C_{free} (see following example). Furthermore, discretization may forbid feasible paths. A solution is to increase the grid resolution. A more efficient hierarchical resolution with data structures such as an octrees, refines the cells close to obstacles.

An alternative discretization with nicer geometric properties relies on triangulation techniques. For a 2D space and polygonal obstacles, the set of vertices of obstacles can be taken as the basis of the decomposition of C_{free} into triangles. Each triangle has 0, 1 or 2 faces adjacent to obstacles. The other 'free' faces define the adjacency graph of the discretized C_{free}. A set of configuration points on each free face and each triangle define the vertices of the graph. Note that by construction, each triangle lies in C_{free}; no collision checking is needed.

There are many triangulation methods for a finite set P of points that are the vertices of the convex polygonal obstacles. The well-known *Delaunay triangulation* is such that no point lies inside any circumcircle of any triangle.[4] This triangulation of P maximizes the minimum of all the angles of the triangles. The centers of the circumcircles of all the triangles are the vertices of the *Voronoi diagram* of P, which is the dual graph of Delaunay triangulation of P. Delaunay triangulation and Voronoi diagrams, generalized to 3D, play an important role in graphics and computer vision, where they are extensively used with quite efficient algorithms. However, these as well as other space discretization techniques do not scale up very

[4]The circumcircle is the circle that passes through the triangle's three vertices.

well to high dimensional spaces and complex environments. They are not widely used for planning the movements of robots with dozens of *dof*.

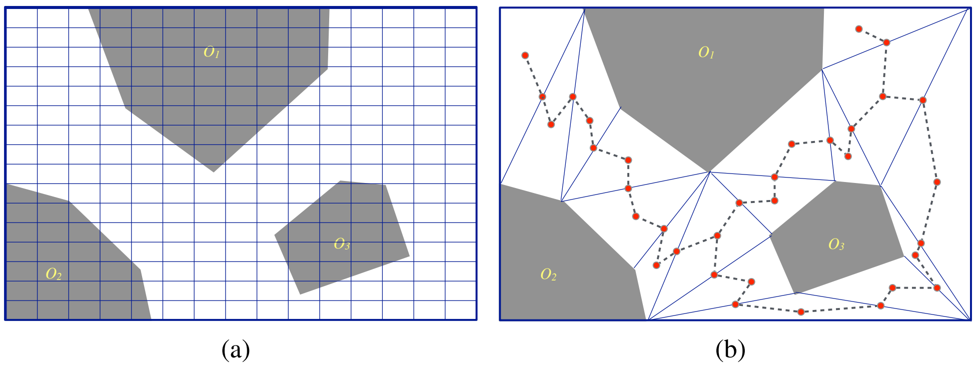


Figure 21.3. Discretization of a simple 2D space with: (a) grid discretization; (b) a triangulation and a corresponding roadmap.

Example 21.5. Consider the 2D space of Example 21.3. Its grid discretization in Figure 21.3(a) is such that the four corners of the cell intersecting the lower edge of obstacle o_1 are all in C_{free}, but the entire cell is not. As in Example 21.4, assuming larger obstacles and/or a rougher grid resolution would make connected areas of this space unconnected through free cells.

A triangulation of this space is illustrated in Figure 21.3(b), together with a corresponding roadmap through the centers and free edges of the triangles (see Exercise 21.3) □

21.1.7 Planning with Movable Obstacles

Motion planning may fail when there is no path to the goal possibly because of obstacles that a robot may remove. Let us assume that some obstacles are labeled as movable and the robot has a function Remove(o) to get rid of a movable obstacle o to which there is a free path. Manipulation planning is required to handle objects to remove them (see Section 21.2). Possibly, several objects may need to be removed in order to dispose of the obstacles to the goal.

Let Mo be the set of movable objects in $\mathcal{W}$, C^{Mo} the configuration space *without* all the objects in Mo, and C^{Mo}_{free} the corresponding free configuration space, i.e., taking into account only unmovable obstacles.

When a motion planner such as Incremental-RRT fails to find a path to the goal, the procedure NAMO seeks a path that can be opened by removing obstacles. It uses a motion planner MP (e.g., RRT or vPRM) on the configuration space C^{Mo} without obstacles. The planner is parameterized such as to return a dense probabilistic roadmap or tree with several paths in C^{Mo} to the goal configuration $\boldsymbol{\theta}_g$. If none is found, NAMO fails. Otherwise it processes all the paths to $\boldsymbol{\theta}_g$ in the returned roadmap $\mathcal{G}$, to find for each one the list of obstacles in Mo which intersect that path. It selects the path with a minimum number of obstacles. It removes these obstacles in the order

```
NAMO($Mo, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g$)
    $\sigma \leftarrow$ Incremental-RRT($\boldsymbol{\theta}_0, \boldsymbol{\theta}_g, n, \eta, p, \Delta n, \rho_\eta, \Delta p$)
    if $\sigma \neq nil$ then return $\sigma$
    $\mathcal{G} \leftarrow$ MP($C^{Mo}, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g$)
    if $\mathcal{G}$ = nil then return nil
    $lpaths \leftarrow \{\langle \boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_g \rangle \in \mathcal{G}\}$
    foreach $\sigma \in lpaths$ do
        $lobs(\sigma) \leftarrow \{o \in Mo \mid o$ intersect with $\sigma\}$
    $\sigma \leftarrow$ argmin$\{lobs(\sigma)\}$
    foreach $o \in lobs(\sigma)$ in the order of intersection with $\sigma$ do
        Remove($o$)
    return $\sigma$
```

Algorithm 21.8. NAMO, a motion planning procedure

in which they are intersected in the chosen path in $\sigma \in lpaths$. There is a free path in C_{free} to the first obstacle in $lobs(\sigma)$, which hence it can be removed, opening a free path to the second obstacle, etc. The returned σ is a path in C_{free} to $\boldsymbol{\theta}_g$.

The assumption that accessible movable objects can be removed may not hold in densely cluttered space. There may not be enough space to dispose of obstacles, or the space required for removing a big obstacle requires the removal of other objects that do not interfere with the path to $\boldsymbol{\theta}_g$. We'll discuss this case in Section 21.3.

21.2 Manipulation planning

We consider here manipulation planning problems that involve only grasping, carrying and ungrasping actions.[5]. The section details the manipulation space and manipulation graph introduced in Section 20.4. It then present a manipulation planning algorithm, first for the case of a single manipulable object, then extended to multiple objects.

21.2.1 Search Space

Recall that the *manipulation space* for an object o in a pose $\mathbf{q}$ and a robot $\mathfrak{R}$ in a configuration $\boldsymbol{\theta}$ is a set of pairs $(\boldsymbol{\theta}, \mathbf{q})$, defined as:

$$\mathcal{M}^o = \{(\boldsymbol{\theta}, \mathbf{q}) \mid \boldsymbol{\theta} \in C_{free} \ \wedge \ (\mathbf{q} \in Q^o_{sta} \ \vee \ (\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp})\}, \text{where}$$

- Q^o_{sta}, as defined in Section 20.4.2, is the set of stable poses of o that do not collide with obstacles, and
- Q^o_{grasp} is the set of pairs $(\boldsymbol{\theta}, \mathbf{q})$ in which a grasp of o is feasible.

We assume that both Q^o_{sta} and Q^o_{grasp} are not empty. However, on a given support, such as a clustered table, there may be no free stable pose for o. Similarly, there may

[5]More would be needed to address the rich repertoire of manipulation actions, e.g., pushing, rolling, switching, flipping, pivoting, turning, screwing, throwing, etc.

be no free grasping configuration $\boldsymbol{\theta}$ for given stable pose $\mathbf{q}$. When there is a pair $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$ and $\boldsymbol{\theta} \in C_{free}$ then the robot in configuration $\boldsymbol{\theta}$ can grasp o in pose $\mathbf{q}$. For such a pair, the transformation $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}) = \mathbf{q}$ puts the end effector in a grasp position.

Once o is grasped, the link between $\mathfrak{R}$ and o becomes rigid: a motion of $\mathfrak{R}$ from $\boldsymbol{\theta}$ to any free $\boldsymbol{\theta}'$ moves o from $\mathbf{q}$ to a pose $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}')$. To ungrasp o, $\mathbf{q}'$ should be in Q^o_{sta}. Given $\boldsymbol{\theta}'$ and $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$, the pose $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}')$ is uniquely defined. But given a target pose $\mathbf{q}'$ there may or may not be a feasible configuration $\boldsymbol{\theta}'$ that allows ungrasping o in $\mathbf{q}'$: the transformation $\mathcal{H}^{-1}_{\boldsymbol{\theta},\mathbf{q}}(\mathbf{q}')$ is an inverse kinematic problem that may not have a solution (see p. 452). In such a case, o may be ungrasped in another pose $\mathbf{q}''$, from which we can see a regrasp that allow the target pose $\mathbf{q}'$ to be reached.

In summary, the manipulation space $\mathcal{M}^o$ characterizes the pairs $(\boldsymbol{\theta}, \mathbf{q})$ where grasping and ungrasping can take place and where $\mathbf{q} \in Q^o_{sta}$ and $(\boldsymbol{\theta}, \mathbf{q}) \in Q^o_{grasp}$. $\mathcal{M}^o$ is a continuous space. The two sets Q^o_{sta} and Q^o_{grasp} may be approximated by discretization. Alternatively, they can be defined with specific procedures to test if o is stable in $\mathbf{q}$ and if a grasp is feasible in $(\boldsymbol{\theta}, \mathbf{q})$. This makes it possible to sample pairs $(\boldsymbol{\theta}, \mathbf{q})$ and check for collision, stability and grasping constraints.

Manipulation planning problems. A *single-object manipulation planning problem* is a tuple $(\mathcal{W}, \mathfrak{R}, o, \boldsymbol{\theta}_0, \boldsymbol{\theta}_g, \mathbf{q}_0, \mathbf{q}_g)$ where $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$ are the initial and final robot configurations in C_{free}, and $\mathbf{q}_0$ and $\mathbf{q}_g$ are two *stable* and *graspable* initial and final poses of the object o.

A solution to a manipulation planning problem is a finite sequence of pairs in $\mathcal{M}^o$: $\langle(\boldsymbol{\theta}_0, \mathbf{q}_0), \ldots, (\boldsymbol{\theta}_n, \mathbf{q}_n)\rangle$ such that $\boldsymbol{\theta}_n = \boldsymbol{\theta}_g, \mathbf{q}_n = \mathbf{q}_g$, for $1 \leq i \leq n$ there is a path in the free configuration space $\wp_{[\boldsymbol{\theta}_{i-1},\boldsymbol{\theta}_i]}$, and either

(i) object o remains immobile while the robot moves to a configuration $\boldsymbol{\theta}_i$ where it can grasp o, that is $\mathbf{q}_i = \mathbf{q}_{i-1}$ and $(\boldsymbol{\theta}_i, \mathbf{q}_i) \in Q^o_{grasp}$, or

(ii) object o remains carried in the same grasp and moves with the robot from $\boldsymbol{\theta}_{i-1}$ to $\boldsymbol{\theta}_i$, where it is ungrasped in a stable pose $\mathbf{q}_i$, that is $\mathbf{q}_i = \mathcal{H}_{\boldsymbol{\theta}_{i-1},\mathbf{q}_{i-1}}(\boldsymbol{\theta}_i)$ and $\mathbf{q}_i \in Q^o_{sta}$.

Note that the specification of a manipulation planning problem may not constrain the final robot configuration: $\boldsymbol{\theta}_n$ can be any point in C_{free}. In practice, one would like $\mathfrak{R}$ to simply move away after ungrasping o in $\mathbf{q}_g$. Note also that there should exist feasible grasps $(\boldsymbol{\theta}, \mathbf{q}_0)$ and $(\boldsymbol{\theta}', \mathbf{q}_g)$ in Q^o_{grasp} that allow to the initial grasp and final ungrasp of o in $\mathbf{q}_0$ and $\mathbf{q}_g$; otherwise the problem has no solution.

It is important to remark that a grasp or an ungrasp action change the robot configuration space C and hence C_{free}. The path $\wp_{[\boldsymbol{\theta}_{i-1},\boldsymbol{\theta}_i]}$ is required to be free in the *current* configuration space. However, these changes in C are local. In a grasp, the changes affect the shape of the end-effector extended with o in the grasped pose $\mathbf{q}_{\boldsymbol{\theta}}(\mathit{eff})$, and the space previously occupied by o. Symmetrically for an ungrasp action. The update of C can be computed incrementally.

Manipulation graphs. A manipulation graph is a directed graph $\mathcal{G}^o = (V^o, E^o)$ such that $V^o \subset \mathcal{M}^o$. E^o has two types of edges, corresponding to the above two cases

(i) and (ii) (see Figure 21.4), that is:

(i) a *transit* edge $((\boldsymbol{\theta}, \mathbf{q}), (\boldsymbol{\theta}', \mathbf{q}))$ represents a path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ in C_{free} in which only $\mathfrak{R}$ moves; o remains immobile at a pose $\mathbf{q} \in Q^o_{sta}$, and

(ii) a *carry* edge $((\boldsymbol{\theta}, \mathbf{q}), (\boldsymbol{\theta}', \mathbf{q}'))$ represents a path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ in C_{free} in which $\mathfrak{R}$ carries o, which moves along this path to a pose $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}')$.

Every outgoing edge from a node $(\boldsymbol{\theta}, \mathbf{q}) \notin Q^o_{grasp}$ must be a transit edge. A carry edge must necessarily connect two nodes $(\boldsymbol{\theta}, \mathbf{q})$ and $(\boldsymbol{\theta}', \mathbf{q}')$ in Q^o_{grasp} such that $\mathbf{q}' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}') \in Q^o_{sta}$. A transit edge for a regrasp action (as in Example 20.8) starts in a state where $\mathfrak{R}$ ungrasps o in a pose $\mathbf{q}$; the edge ends in a state where $\mathfrak{R}$ regrasps o from another pose $\mathbf{q}'$, but the two grasping relations are different: $\mathcal{H}_{\boldsymbol{\theta}',\mathbf{q}'} \neq \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$.

A manipulation planning algorithm performs a search in the manipulation space. It builds a manipulation graph starting at node $(\boldsymbol{\theta}_0, \mathbf{q}_0)$ seeking a path in this graph of *alternate* transit and transfer edges that terminates at a node $(\boldsymbol{\theta}_n, \mathbf{q}_g)$ where o is in its goal pose. The condition on alternate types of edges is simply because two consecutive transit edges $\langle((\boldsymbol{\theta}, \mathbf{q}), (\boldsymbol{\theta}', \mathbf{q})), ((\boldsymbol{\theta}', \mathbf{q}), (\boldsymbol{\theta}'', \mathbf{q}))\rangle$ do not bring much to the plan and can be merged into a single edge: $\langle((\boldsymbol{\theta}, \mathbf{q}), (\boldsymbol{\theta}'', \mathbf{q}))\rangle$. The intermediate point $\boldsymbol{\theta}'$ is integrated in the path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}'']}$. Similarly for two consecutive carry edges in the same grasp. Consequently, we can partition the vertices into two types of nodes: $V^o = V^o_t \cup V^o_c$, where

(i) V^o_t is the set of *t-nodes* from which only *transit* edges are issued, and

(ii) V^o_c is the set of *c-nodes*, from which only *carry* edges are issued.

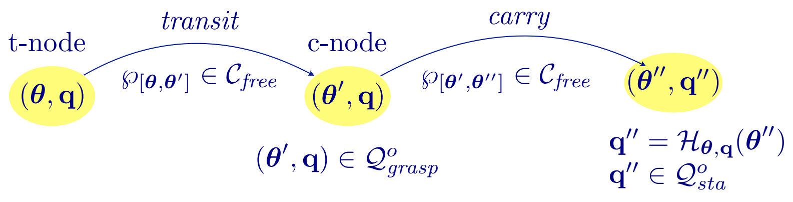

transit
carry
t-node
c-node
$\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']} \in \mathcal{C}_{free}$
$\wp_{[\boldsymbol{\theta}',\boldsymbol{\theta}'']} \in \mathcal{C}_{free}$
$(\boldsymbol{\theta}, \mathbf{q})$
$(\boldsymbol{\theta}', \mathbf{q})$
$(\boldsymbol{\theta}'', \mathbf{q}'')$
$(\boldsymbol{\theta}', \mathbf{q}) \in \mathcal{Q}^o_{grasp}$
$\mathbf{q}'' = \mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}(\boldsymbol{\theta}'')$
$\mathbf{q}'' \in \mathcal{Q}^o_{sta}$

Figure 21.4. Nodes and edges of a manipulation graph.

A simple *pick-and-place* manipulation problem (that is, moving an object between two poses in an uncluttered environment) can be addressed as two consecutive motion planning problems, to plan motions to the pick-up pose and the placement pose. A more complex manipulation, however, may require significant search in $\mathcal{G}_o$, as illustrated next.

21.2.2 Manipulation Planning Algorithm

The single-query manipulation planner ManipPlanner grows a tree of nodes in $\mathcal{M}^o$ using a strategy similar to RRT. At each iteration, it alternatively expands either a t-node or a c-node, which are randomly chosen in V^o with respectively Choose-t or Choose-o. From a t-node it uses Grasping to seek a grasping configuration with a free path to be added to the tree through a transit edge. From a c-node, it uses Ungrasping to seek an ungrasping pose and a configuration with a free path to be added to the tree through a carry edge. It stops when a pair reaches the pose $\mathbf{q}_g$, assuming that the final robot configuration is unconstrained. Otherwise, a final motion step to reach a configuration $\boldsymbol{\theta}_g$ would be needed.

```
ManipPlanner($\boldsymbol{\theta}_0, \mathbf{q}_0, \mathbf{q}_g, n$)
    $V^o \leftarrow \{(\boldsymbol{\theta}_0, \mathbf{q}_0)\}$; $E^o \leftarrow \emptyset$; $flag \leftarrow \top$
    until $(\exists\boldsymbol{\theta} : (\boldsymbol{\theta}, \mathbf{q}_g) \in V^o)$ or $|V^o| = n$ do
        if $flag$ then                                        // expand a t-node
            $(\boldsymbol{\theta}, \mathbf{q}) \leftarrow$ Choose-t($V^o$)
            $(\boldsymbol{\theta}', \mathbf{q}') \leftarrow$ Grasping($\boldsymbol{\theta}, \mathbf{q}, k$)        // here $\mathbf{q}' = \mathbf{q}$
        else                                                  // expand a c-node
            $(\boldsymbol{\theta}, \mathbf{q}) \leftarrow$ Choose-o($V^o$)
            $(\boldsymbol{\theta}', \mathbf{q}') \leftarrow$ Ungrasping($\boldsymbol{\theta}, \mathbf{q}, k$)
        if $(\boldsymbol{\theta}', \mathbf{q}') \neq$ nil then
            $V^o \leftarrow V^o \cup \{(\boldsymbol{\theta}', \mathbf{q}')\}$
            $E^o \leftarrow E^o \cup \{(\boldsymbol{\theta}, \mathbf{q}), (\boldsymbol{\theta}', \mathbf{q}')\}$
            $flag \leftarrow \neg flag$
    if $\exists\boldsymbol{\theta} : (\boldsymbol{\theta}, \mathbf{q}_g) \in V^o$ then return $\mathcal{G}^o = (V^o, E^o)$
    else return failure
```

Algorithm 21.9. ManipPlanner, a single-query manipulation planning algorithm

From a t-node $(\boldsymbol{\theta}, \mathbf{q})$, a call to Grasping$(\boldsymbol{\theta}, \mathbf{q}, k)$ fails if there is no grasping configuration for o in the pose $\mathbf{q}$. Otherwise, Grasping samples a free configuration $\boldsymbol{\theta}'$ in Q^o_{grasp} with respect to $\mathbf{q}$, i.e., such that $(\boldsymbol{\theta}', \mathbf{q}) \in Q^o_{grasp}$. Since $(\boldsymbol{\theta}, \mathbf{q})$ is a t-node, the robot moves empty handed along the transit edge to the new configuration $\boldsymbol{\theta}'$. The grasp takes place in the pose $\mathbf{q}$. The function FreePath calls a motion planner and returns a path in C_{free} from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$ if one is found or nil otherwise. If a path is found, the procedure returns $(\boldsymbol{\theta}', \mathbf{q})$ (which we will denote as $(\boldsymbol{\theta}', \mathbf{q}')$, with $\mathbf{q}' = \mathbf{q}$, for ease of the follow-up notation): this pair is reachable with a feasible grasp of o in $\mathbf{q}$. Otherwise a new grasp $\boldsymbol{\theta}'$ is sampled. After k unsuccessful trials, Grasping returns nil. Another t-node $(\boldsymbol{\theta}, \mathbf{q}) \in V^o$ will be tried by ManipPlanner.

```
Grasping($\boldsymbol{\theta}, \mathbf{q}, k$)
    if $\{\boldsymbol{\theta}' | (\boldsymbol{\theta}', \mathbf{q}) \in Q^o_{grasp}\} = \emptyset$ then return nil
    for $k$ times do
        $\boldsymbol{\theta}' \leftarrow$ Sample-grasp($Q^o_{grasp}, \mathbf{q}$)        // $(\boldsymbol{\theta}', \mathbf{q}) \in Q^o_{grasp}$
        if FreePath($\boldsymbol{\theta}, \boldsymbol{\theta}'$) then return $(\boldsymbol{\theta}', \mathbf{q})$
    return nil
```

Algorithm 21.10. Grasping, sampling a feasible and reachable grasping configuration.

From a c-node $(\boldsymbol{\theta}, \mathbf{q})$, Ungrasping seeks a stable pose $\mathbf{q}'$ in which to put o. Since o is already held in the end-effector, $\mathcal{H}_{\theta,\mathbf{q}}$ is given from $(\boldsymbol{\theta}, \mathbf{q})$. However, its inverse $\mathcal{H}^{-1}_{\theta,\mathbf{q}}(\mathbf{q}')$ may or may not have a solution $\boldsymbol{\theta}'$ to ungrasp o in a sampled pose $\mathbf{q}'$. This sampling can be driven towards the goal pose $\mathbf{q}' = \mathbf{q}_g$. If there is a path in $\mathcal{C}_{free}$ from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$ where o can be ungrasped, the procedure returns the pair $(\boldsymbol{\theta}', \mathbf{q}')$. Otherwise news poses are sampled until reaching the maximum number of allowed trials.

```
Ungrasping($\boldsymbol{\theta}, \mathbf{q}, k$)
    for $k$ times do
        $\mathbf{q}' \leftarrow$ Sample-pose($Q^o_{stable}$)
        if ($\boldsymbol{\theta}' \leftarrow \mathcal{H}^{-1}_{\theta,\mathbf{q}}(\mathbf{q}')$) exists then
            if FreePath($\boldsymbol{\theta}, \boldsymbol{\theta}'$) then return ($\boldsymbol{\theta}', \mathbf{q}'$)
    return nil
```

Algorithm 21.11. Sampling a stable pose $\mathbf{q}'$ towards the goal and a corresponding feasible ungrasping configuration $\boldsymbol{\theta}'$.

Note that the configuration space C needs to be incrementally updated at each expanded node $(\boldsymbol{\theta}, \mathbf{q})$ to account for the current pose of o. To amortize these updates, it can be worthwhile to expand a node through several branching edges.

Searching in the manipulation space is computationally more demanding than searching in the configuration space. It adds an additional dimension of poses coupled to configurations, with sampling in Q^o_{grasp} and Q^o_{stable}. It also involves at each expansion step an RRT motion planning for a free path. However $\mathcal{G}^o$ is significantly a smaller graph than a roadmap or an RRT tree.

The algorithm ManipPlanner can benefit from several heuristics, e.g.,

- Sample-grasp should seek least-constrained free configurations in the range of allowable grasps for o in pose $\mathbf{q}$,
- Sample-pose should seek feasible stable poses close to the goal, possibly $\mathbf{q}' = \mathbf{q}_g$ with $\boldsymbol{\theta}' \in \mathcal{C}_{free}$, or far from obstacles allowing further re-grasps.

Example 21.6. Consider a simple manipulation problem where a forklift-like robot can hold and manipulate 2D objects. The task illustrated in Figure 21.5 is to displace the orange box from pose $\mathbf{q}_0$ to $\mathbf{q}_g$. There are only four grasp positions in Q^o_{grasp}, denoted a, b, o, d. The root of the search tree corresponds to $\mathbf{q}_0$ with the robot at its initial configuration. Because of obstacles, Grasping can generate only two successor nodes to the root, corresponding to the a and b configurations. From these nodes Ungrasping may generate several poses $\mathbf{q}_1$ to $\mathbf{q}_4$ of the box in the free space of the room. From one of these nodes, say $(a, \mathbf{q}_2)$ the robot may reach a grasp $(o, \mathbf{q}_2)$, from which it can move the box to the goal. Alternatively, the node $(b, \mathbf{q}_2)$ or even $(b, \mathbf{q}_0)$ may also allow reaching the goal with a pose b. Along a transfer edge (in blue) the object pose does not change; along a carry edge (in red) the grasp does not change. □

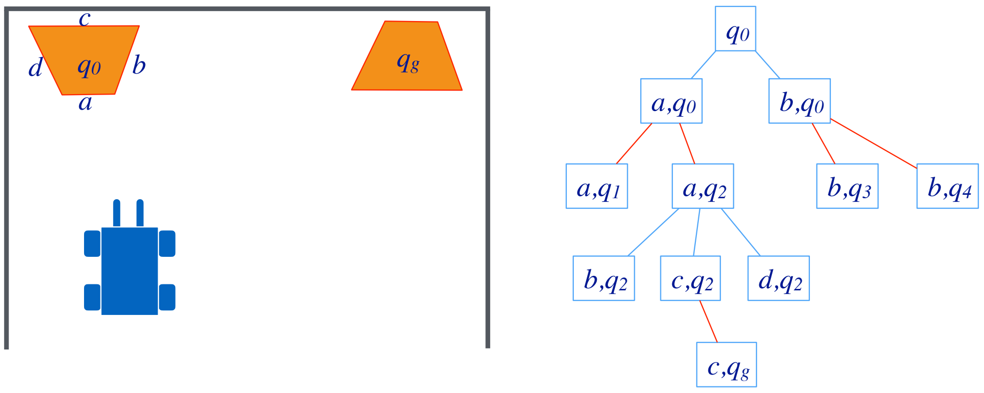


Figure 21.5. A forklift-like robot has to displace a 2D box (in orange) from $\mathbf{q}_0$ to $\mathbf{q}_g$; a search tree for this task in the manipulation space (transfer edges are blue, carry edges are red).

Example 21.7. Figure 21.6 illustrates a more complex single-object manipulation task: a robot arm has to extract a ruler-like object (in grey) from under a cage. Several transit and carry movements are needed in the narrow area allowed by the cage before the tip of the ruler extends out of the cage and can be grasped to fully extract the object. □

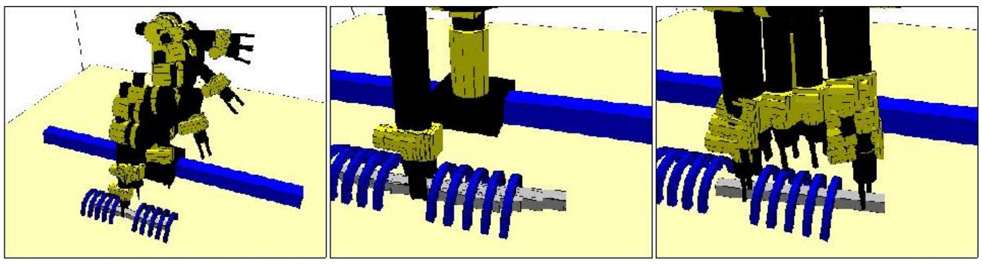


Figure 21.6. Starting from the left image, the robot repeatedly grasps the ruler in the middle interval allowed by the cage, and moves it slightly several times until the ruler is graspable outside of the cage, to extract it from this constrained space (from [1023]).

Manipulation of multiple objects. A single-object manipulation-planning problem is fairly constrained. Even in problems involving only grasping, carrying and ungrasping actions, one would like to be able to temporarily move other objects out of the way to allow for a grasp or a free path to a destination pose, e.g., if we had a movable obstacle between the robot and the orange object in Example 21.6. Handling tasks such as the rearrangement of multiple objects is not equivalent in general to a sequence of a single-object manipulation problems.

The manipulation space for multiple objects is a set of tuples $(\boldsymbol{\theta}, \mathbf{q}_{o_1}, \dots, \mathbf{q}_{o_k})$ giving the configuration $\boldsymbol{\theta}$ and the poses of all movable objects in the domain, each pose being stable and/or in a grasp. The manipulation graph needs to be extended to vertices linking such tuples with transit and carry edges. A carry edge is related to the same object, while a transit edge connects an ungrasp of some o towards a grasp of possibly another o'.

The principles of the ManipPlanner algorithm with procedures Grasping and Ungrasping are applicable, with a few extensions, to multiple-object manipulation problems. We need to distinguish the case where Grasping finds a feasible grasp configuration $\boldsymbol{\theta}'$ but no free path $\wp[\boldsymbol{\theta}, \boldsymbol{\theta}']$ exists; this is to allow for the removal of obstacles to $\boldsymbol{\theta}'$. Similarly for Ungrasping (see Example 21.10). We also need to handle the choice of which object to manipulate at each stage. The resulting tree would grow significantly in complexity. This, among other reasons, advocates for addressing extended manipulation planning with a combined task- and motion-planning approach.

Another extension would consider several end-effectors for moving the objects. A dual-arm mobile robot such as Justin (Figure 20.6(a)) allows for more flexible manipulations, such as regrasping from one arm to the other. Similarly, several robots allow for addressing different accessibility and grasping constraints and for handing an object from one robot to another. Here too the ManipPlanner framework can in principle be extended to several arms and/or robots. These extensions follow the same principles if we restrict each grasp to a single end-effector. Additional problems need to be solved if we consider dual-arm grasps, e.g., carrying a tray, which involve closed kinematic chains. These extensions come at the cost of a more complex search.

Finally, the RRT-like approach of ManipPlanner, which can be seen as a step towards the integration of motion, manipulation and task planning, may not be the best one for all types of manipulation tasks. In particular, a series of manipulation-planning tasks in the same unchanging environment may call for a multiple-query PRM-like approach (see Section 21.4.2).

21.3 Task, Motion and Manipulation Planning

In the two previous sections, we focused on searching for a metric path between two configurations $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_g$, or between two pairs of configurations-poses $(\boldsymbol{\theta}_0, \mathbf{q}_0)$ and $(\boldsymbol{\theta}_g, \mathbf{q}_g)$. The available actions, moving, grasping and ungrasping, were defined solely with metric operational models (as discussed in Chapter 20). The motion-manipulation planner did not use these actions *per se*; they were implicit in edges in the manipulation graph. There was no need to specify a repertoire of available actions and refer in a planning algorithm to the actions chosen in a plan, as we did for task planning. We did not have abstract actions with causal precondition-effect relations. In combined Task and Motion Planning (TAMP), we would like tackle more complex tasks, requiring motion and other types of actions, with a planner that can rely on named available actions and their models, including kinematics as well as their effects on state variables of interest. Let us motivate with an example.

Example 21.8. Consider a mobile robot, called Butler, that has to set up the table. For that, it will have to clear and clean it, open and close cabinets and drawers, take and fixe a tablecloth, take dishes, glasses and silverware, carry and place them at their appropriate places, and push chairs. Its repertoire of actions is {clear, clean, open, close, take, fixe, carry, place, push}. Similarly, consider a robot called Storekeeper that services a warehouse. It has to handle boxes, take them from and put them into shelves, stack and unstack them, sort, pack and unpack them, remove obstacles, navigate the warehouse, open doors and operate elevators. Its repertoire is: {handle, take, put, stack, unstack, sort, pack, unpack, remove, navigate, open, operate}.

To accomplish these high level tasks, it is helpful to use descriptive models of actions, their preconditions and effects, as well as on their metric motion and manipulation models. □

Almost all of the actions to be performed by the Butler or Storekeeper robots involve movements (possible exceptions are, e.g., operating elevators with remote-control signals). However, the motion and manipulation planners presented in the two previous sections are not designed for planning these complex tasks. We need to adequately model several distinct tasks and/or primitive actions. For that, we have to use both the symbolic task planning representation seen in the previous parts of the book and the metric representation described in this part.

A simple idea makes a parallel between

- combined task and motion planning, our topic here, and
- interleaved task planning and acting, discussed in several parts of the book.

The intuition is to do task planning first, then motion-manipulation planning, i.e., to plan for abstract actions, and postpone the refinement of planned actions into movements at a lower level, possibly at acting time. The approach finds a task plan, ignoring metric relations and constraints, then, at acting time, it checks these constraints and seeks with a metric planner needed movements (see Figure 21.7).

This approach works for sparse, unconstrained environments and simple tasks, but it can be quite inefficient for demanding activities. This is because the task planner does not reason about space and kinematic constraints; it may plan actions that appear causally valid at an abstract level but are metrically unfeasible. For example, the Storekeeper robot may plan to stack boxes in a place that obstructs the rest of the plan. Backtracking in the real world is seldom a good idea and synonymous of poor planning. In critical domains, this might even put the robot in a dead end.

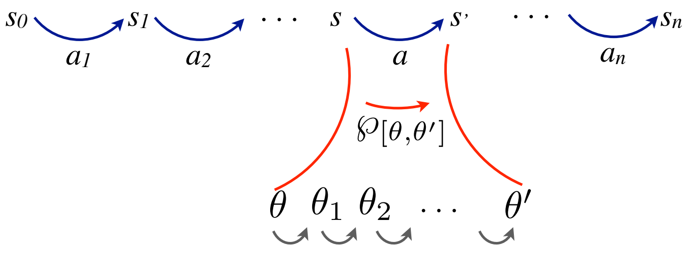


Figure 21.7. A simplified view which considers a task plan as a sequence of abstract state transitions, each step of which is to be refined later into a sequence of configurations (and object poses) in a metric path $\wp_{[\theta_0,\theta_g]}$.

The main challenge in integrating task and motion planning is how to efficiently combine two heterogeneous representations, how to map back and forth the high-level task descriptions and the low-level metric models. Approaches addressing this challenge can be (roughly) categorized in three strategies:

(i) *Motion-constrained task planning*: a task planner synthesizes a symbolic plan whose metric constraints, such as the existence of free paths and reachable grasps, are checked incrementally for each candidate sub-plan while planning.

(ii) *Task-guided motion planning*: a motion planner searches for a sequence of metric plans in a sequence of configuration spaces, using task-planning methods as guidance for incrementally building this sequence.

(iii) *Interleaved task and motion planning*: a single planner uses an action representation that combines operational and descriptive models with metric and causal relations; it explores a search space that integrates the high-level task constraints and low-level metric constraints.

The first two strategies *decouple* in some way motion planning and task planning, with a priority to task planning for (i), and to motion planning for (ii). Strategy (iii) integrates the heterogenous components of the problem in unified view. Each strategy has advantages in particular problems. For example, one would use (ii) for climbing a ladder since movement is the dominant part.

For the three strategies, the metric part is computationally more demanding than the symbolic part, often by more than an order of magnitude. Checking the feasibility of movement and grasp constraints over every step of a combinatoric search is not a good option. Several techniques for attacking this problem with one of these three strategies have been proposed (surveyed in Section 21.4.3).

In the reminder of this section, we focus on strategy *(iii)* and discuss how to combine the metric and symbolic representations, and how to merge their techniques in interleaved task and motion planning.

21.3.1 Representation and Search Space for TAMP

Before defining TAMP problems and TAMP plans let us discuss the representation needed in TAMP, which has to combine:

- the metric representation presented in this part, which includes geometric, kinematic and dynamic models of the environment and the robot, and
- the symbolic representation used in previous parts of the book.

TAMP uses metric state variables and symbolic state variables. A metric state variable x_i, such as a robot configuration or an object pose, has a metric range $\mathbb{R}^{\alpha_i}$, endowed with a metric function (see Appendix B.1) that a planner takes advantage of.[6] A symbolic state variable x_j has a discrete range D_j; it refers to the symbolic labels of objects, containers, support surfaces, locations, robots, and their properties. Let us illustrate the hybrid representation with a first example.

[6]Metric state variables are generally but not necessarily continuous:. A discrete range may have a metric function, e.g., the battery level or the container weight can be metric variables over the integers.

Example 21.9. Consider the DWR domain in Example 14.1, in which robots can take, move and put containers (for example, with forklifts as in Figure 2.1). In this domain, we have finite sets of object variables, such as Robots, Locations, Containers, and Supports. To these, we add two metric sets:

- Configurations $\subset \mathbb{R}^4$: set of robot configurations, with 3 *dof* on the plane and one *dof* for its fork position, and
- Poses $\subset \mathbb{R}^4$: the set of container poses with 4 *dof* on a planar surface.

We had symbolic state variables such as $\mathsf{loc}(r) \in$ Locations, $\mathsf{place}(o) \in$ Robots $\cup$ Locations, or $\mathsf{cargo}(r) \in$ Containers $\cup$ {empty}, for $r \in$ Robots, and $o \in$ Containers. We add the following metric state variables:

- $\mathsf{config}(r) \in$ Configurations: the current configuration of a robot r,
- $\mathsf{pose}(o) \in$ Poses: the current pose of a container o.

The schema take specifies that an empty robot takes a container in its location; put says that a robot puts a container it carries it in its location.

```
take(r, o, θ, q, l);
        pre: $\mathsf{config}(r) = \boldsymbol{\theta}, \mathsf{pose}(o) = \mathbf{q}$,
             $\mathsf{loc}(r) = l, \mathsf{place}(o) = l, \mathsf{cargo}(r) = \mathsf{nil}$
     sample: $(\boldsymbol{\theta}' \overset{\cdot}{\leftarrow} \mathsf{Grasping}(r, o, \boldsymbol{\theta}, \mathbf{q})) \neq \mathsf{nil}$,          (1)
        eff: $\mathsf{config}(r) \leftarrow \boldsymbol{\theta}', \mathsf{cargo}(r) \leftarrow o$
             $\mathsf{pose}(o) \leftarrow \mathcal{H}_{\theta',\mathbf{q}}(\boldsymbol{\theta}')$
place(r, o, θ, q, l);
        pre: $\mathsf{config}(r) = \boldsymbol{\theta}, \mathsf{pose}(o) = \mathbf{q}$,
             $\mathsf{loc}(r) = l, \mathsf{place}(o) = r, \mathsf{cargo}(r) = o$
     sample: $(\boldsymbol{\theta}' \overset{\cdot}{\leftarrow} \mathsf{Ungrasping}(r, o, \boldsymbol{\theta}, \mathbf{q})) \neq \mathsf{nil}$,          (2)
        eff: $\mathsf{config}(r) \leftarrow \boldsymbol{\theta}', \mathsf{holding}(r) \leftarrow \mathsf{nil}, \mathsf{on}(o) \leftarrow p$
             $\mathsf{pose}(o) \leftarrow \mathcal{H}_{\theta,\mathbf{q}}(\boldsymbol{\theta}')$
```

The action take is conditioned on the existence of a graspable configuration $\boldsymbol{\theta}'$ to grasp o in its current pose $\mathbf{q}$, such that $(\boldsymbol{\theta}', \mathbf{q}) \in Q^o_{grasp}$ and there is a free path for r from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$ (line 1)). Similarly, the action put is conditioned on the existence of a stable configuration $\boldsymbol{\theta}'$ to ungrasp o such that $\mathcal{H}_{\theta,\mathbf{q}}(\boldsymbol{\theta}') \in Q^o_{sta}$ and there is a free path for r from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$ (line 2)).[7] While the other preconditions result from matching state variables with their values in the current state, these are conditioned on the existence of *sampled* configuration $\boldsymbol{\theta}'$ and associated free path as computed by the functions Grasping and Ungrasping. An instance of an action deterministically either holds or does not hold in a given state with respect to its normal preconditions, but it nondeterministically depend on the sampled variables, which have to be dealt with through the iterative sampling procedures seen earlier.

For clarity, preconditions with sampled variables are separated with a syntactic marker 'sample'. The sampling operation is denoted as $\overset{\cdot}{\leftarrow}$ to distinguish it from the usual assignments in the effects. Note that the new pose(o) also results from a numeric computation by $\mathcal{H}_{\theta,\mathbf{q}}$ relying on the sampling of $\boldsymbol{\theta}'$. □

[7] We may constrain Ungrasping to find the target pose in some place or location.

Let us denote the joint ranges of metric and symbolic state variables as:

- $\Omega \subseteq \prod_i \mathbb{R}^{\alpha_i}$, for i over all metric state variables x_i, and
- $S \subseteq \prod_j D_j$, for j over all symbolic state variables x_j.

The search space of a TAMP problem is the Cartesian product $\Omega \times S$. Clearly, it is not finite. A state can be written as a pair (ω, s), where

- $\omega = (\boldsymbol{\theta}, \mathbf{q}_{o_1}, \ldots, \mathbf{q}_{o_k}) \in \Omega$, is the metric component of the state; it gives the robot configuration and the poses of movable objects in $\mathcal{W}$; and
- $s \in S$, is the symbolic component, i.e., a conjunction of pairs (state variable, value) for all symbolic state variables.

A TAMP planner explores the hybrid search space $\Omega \times S$ by combining one or several searches through Ω and S relying on their distinct mathematical structure. Let a be an action in a TAMP domain applicable in a state (ω, s). The effects of a are modeled as a deterministic mapping[8] from (ω, s) to a state (ω', s'), which can be decomposed with two transition functions φ and γ:

$$a : (\omega, s) \longrightarrow (\omega', s') \text{ with } \begin{cases} \omega' = \varphi(\omega, s, a) \\ s' = \gamma(\omega, s, a) \end{cases}$$

Note that φ is a function s as well as of ω; similarly γ is a function ω as well as s. The two transition levels in Figure 21.7 are intricately coupled. The metric and symbolic components can rarely be managed with two decoupled transition functions. The issue in TAMP is how dependencies within and between symbolic and metric state variables are handled.

In general, the state variables of a domain are not independent. This is already the case for dependencies within symbolic variables in task planning, e.g., in the DWR domain (Example 2.1) there are dependencies between cargo(r) and pos(o) or between loc(r) and occupied(d). Similarly, in motion and manipulation planning, there are dependencies within metric variables due to invariant properties of $\mathcal{W}$, to geometric, kinematic, or physical constraints, e.g., collision or stability (see Section 20.2). Moreover, in TAMP there dependencies between the two types of variables, e.g., in Example 21.9 cargo(r), pose(o) and on(o) are dependent.

Dependency constraints in task planning are often handled *implicitly*: assuming that the initial state meets the constraints, the action models restrict the set of reachable states to only those that also meet the contraints. In motion and manipulation planning, explicit means have been introduced to handle these constraints, e.g., the transformation $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$ and the sets Q^o_{grasp} and Q^o_{stable} of feasible grasps and stable poses of an object o. These, as well as additional means are also needed in TAMP to explicitly handle the constraints between symbolic and metric variables . For example, an action that transport an object, say box3 needs to compute its new new metric pose(box3), as well as to find or check out a symbolic attribute such as on(box3)=shelf2.

In summary, action models in a TAMP planner have to specify how to compute γ and φ while handling the constraints. This is done with:

[8]Planners that handle the inaccuracy of motion and manipulation with nondeterministic models are discussed in Section 21.4.3

(i) *Local metric methods* for computing $\boldsymbol{\omega}'$ from $\boldsymbol{\omega}, s$ and a. These are the procedures seen earlier to compute the kinematic reachability, collision detection, graspability and stability constraints.

(ii) *Explicitly asserted symbolic* changes: s' results from $s, \boldsymbol{\omega}$ and a with explicitly stated changes in the effects of an action a for some state variables, the other state variables remain unchanged.

(iii) *Functionally defined changes*: some state variables in $(\boldsymbol{\omega}', s')$ are defined as functions of other state variables in previous and/or current state, and computed by domain-specific procedures. These functions are implemented as programs conveniently grouped in a *geometric reasoner*, i.e., a library which computes state variables such as on(box3)=shelf2, in(o1)=box2, touching(o2,o3), as well as functions such as FreePath$(r, \boldsymbol{\theta}, \boldsymbol{\theta}')$, Stable$(o, \mathbf{q})$, Graspable$(r, o, \boldsymbol{\theta}, \mathbf{q})$.

Finally, let us notice that an action a that involves a movement between two configurations $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ has to necessarily check the existence of a path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ in C_{free}, which needs to be kept as part of the plan containing a.

TAMP planning domains and problems. A task and motion planning domain is a tuple $\Sigma = (\mathcal{W}, \mathfrak{R}, \Omega, S, A)$ where $\mathcal{W}, \mathfrak{R}, \Omega, S$ are as defined above, A is a set of precondition/effect action schemas defining the two state transition functions γ and φ. A task and motion planning problem is a tuple $(\Sigma, \omega_0, s_0, g)$ where (ω_0, s_0) is the initial state and $g \in \Omega \times S$ is a set of goal states. Usually the goal does not specify the precise final poses of objects in space but simply requirements about their locations and places, e.g., in(o1)=box2, on(o2)=table1; similarly for goal configuration of the robot.

A solution to a TAMP problem $(\Sigma, \omega_0, s_0, g)$ is a sequence of action-path pairs: $\pi = \langle(a_1, \wp_{[\boldsymbol{\theta}_0,\boldsymbol{\theta}_1]}), \ldots, (a_k, \wp_{[\boldsymbol{\theta}_{k-1},\boldsymbol{\theta}_k]})\rangle$, corresponding to a sequence of states $\langle(\omega_0, s_0), \ldots, \ldots, (\omega_g, s_g)\rangle$, such that $s_{i+1} = \gamma(\omega_i, s_i, a_i)$, $\omega_{i+1} = \varphi(\omega_i, s_i, a_i)$, (ω_g, s_g) is a goal state, and $\wp_{[\boldsymbol{\theta}_i,\boldsymbol{\theta}_{i+1}]}$ is a path in C_{free} from $\boldsymbol{\theta}_i$ to $\boldsymbol{\theta}_{i+1}$, the respective configurations in $\boldsymbol{\omega}_i$ and $\boldsymbol{\omega}_{i+1}$. If there is no movement in a transition (a static action such as sending a message), then $\boldsymbol{\omega} = \boldsymbol{\omega}'$ and $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']} = \varnothing$.

21.3.2 TAMP with a Forward-Search Planner

In Part I and Part II, we studied several deterministic planning algorithms that generate a sequential plan in a forward order, starting from its first action. These are for example the generative Forward-Search state space planner or TO-HTN-Forward HTN planner. Let us discuss how to extend these planning approaches while interleaving task and motion planning for TAMP problems.

We use precondition/effect action schemas with hybrid metric and symbolic state variables as illustrated in Example 21.9. The preconditions relies on procedures such as FreePath, Grasp-config and Ungrasp-config, to compute by sampling metric state variables that meet metric constraints. The effects specify the updates for the next state in $S \times \Omega$ from the values computed and/or tested in the preconditions, possibly with metric transformation, as $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$. Note that these and other motion/manipulation functions are computationally very expensive. They should be used as sparsely as possible by a TAMP planner, as discussed next.

Sampling is inherent to most of the functions needed for metric preconditions. It entails complexity and complications on a TAMP algorithm. A failure for an action a (e.g., to find a free path, a graspable configuration or a stable pose) might be due not to the actions chosen before a but to sampled values in a and before a. For example, the configuration sampled by Grasp-config to take o may forbid finding later a free path or a stable pose where to put it (e.g., taking a container from its long side instead of the narrow side).

A possible approach would add the sampled variables in the parameters of actions such as to make an action instance explicitly integrate the sampling performed in its preconditons. But, this would not solve the issue of branching the search over symbolic variables as well as over sampled metric variables. Backtracking over the set of possible samples is not an efficient option. Fortunately metric variables range over sets with metric functions that can be leveraged for the guidance of a planner.

```
F-TAMP($\Sigma, \omega_0, s_0, g$)
    $Frontier \leftarrow \{(\langle\rangle, \omega_0, s_0)\}$ ; $Expanded \leftarrow \varnothing$
    $U((\langle\rangle, \omega_0, s_0)) \leftarrow 0$
    while $Frontier \neq \varnothing$ do
1       $(\pi, \omega, s) \leftarrow \mathrm{argmax}_{(\pi_i, \omega_i, s_i) \in Frontier} U(\pi_i)$
        remove $(\pi, \omega, s)$ from $Frontier$ and add it to $Expanded$
2       if postponed variable samplings are s.t. $\omega = \varphi(\omega_0, s_0, \pi)$ then
3           if $(\omega, s)$ satisfies $g$ then return $\pi$
4           foreach $a \in P\text{-}applicable(\omega, s)$ do
5               $Q_0(\omega, s, a) \leftarrow r_s(\omega, s, a)/h(\omega, s, a, g)$
                $s' \leftarrow \gamma(\omega, s, a)$
6               $\omega' \leftarrow P\text{-}instance(\omega, s, a)$
7               if $Q_0(\omega, s, a) > \epsilon$ and $(\omega', s') \notin Frontier \cup Expanded$ then
                    $\pi' \leftarrow \pi \cdot a$
                    $U(\pi') \leftarrow U(\pi) + Q_0(\omega, s, a)$
                    add $(\pi', \omega', s')$ in $Frontier$
    return failure
```

Algorithm 21.12. F-TAMP, a forward-search task and motion planner.

Algorithm F-TAMP is an instance of the Forward-Search-Det schema, augmented with an appropriate handling of sampled metric variables and associated motion and manipulation procedures. *Frontier* is a sorted list of current alternative plans, ordered with respect to utility fonction U:

$$U(\pi) = \sum_{a \text{ in } \pi} Q_0(\omega, s, a), \text{ with}$$
$$Q_0(\omega, s, a) = r_s(\omega, s, a)/h(\omega, s, a, g) \tag{21.1}$$

The action-value function Q_0 for a in (ω, s) takes into account two terms:

- $r_s(\omega, s, a) \in [0, 1]$, the success reward of a in (ω, s)(see Section 15.2.3), and
- $h(\omega, s, a, g) \in \mathbb{R}^+$ a heuristic estimate of how much a is expected to contribute progressing towards g.

The higher is r_s and the lower is h, the more valuable is a for the current search node. If Q_0 is lower than a threshold $\epsilon > 0$ the corresponding action is pruned. Section 22.2 explains how to learn Q_0.

F-TAMP expands the search node maximizing U. Only at this stage the feasibility of the expanded node is fully checked (in Line 2). Since expanding a node branches over possibly many successors, testing all of them with motion and manipulation procedures would be too expensive. F-TAMP postpones this testing until a successor is selected with its expected utility U. At this stage motion and manipulation planners are used to the sample postponed variables needed by the action a and check that the partial plan π feasible. If $\omega \neq \varphi(\omega_0, s_0, \pi)$, then the feasibility test fails; the corresponding node is simply removed from *Frontier*. Otherwise F-TAMP uses *P-applicable* to examine each action a that is *possibly* applicable to (ω, s), i.e., disregarding preconditions that require sampling with motion/manipulation planning. The function $Q_0(\omega, s, a)$ and the state (ω', s') are computed, with the caveat that the sampled variables of a are at this stage missing from ω' (function *P-instance*). Unless pruned (in Line 7) this successor node is added to frontier.

An alternative to the pseudo-code in Algorithm 21.12 is to postpone testing the motion/manipulation feasibility to terminal nodes, when (ω, s) possibly satisfy g (i.e., moving the test in Line 2 as a condition in Line 3). This amount basically to searching for a "skeleton" plan, that meets only the symbolic part of Σ, with the guidance of the utility function U, then testing its metric feasibility once completed, with further node expansions if the test fails.

To sum up, F-TAMP progresses in a forward-search over all possibly applicable actions (Line 4). The symbolic preconditions are taken as necessary but not sufficient condition. It postpones the feasibility testing of the sampled numeric variables to the expansion stage (Line 2), or possibly even to the final stage. This is because the motion/manipulation part is the most costly and has to be used sparingly.

The efficiency of F-TAMP relies on how informative the reward function r_s is. Prior learning can provide a quite precise r_s function that rules out unlikely feasible actions and favors most likely feasible ones, giving a good focus to the search (see Section 22.2). The probabilistic completeness of F-TAMP is conditioned on the action-value function meeting: $Q_0(\omega, s, a) > \epsilon$ whenever a is feasible in (ω, s). This in turn depends mainly on the expected success reward function, which needs to be strictly positive for feasible actions. The heuristic h plays a less critical role; it may rely solely on the symbolic part of Σ and use the classical techniques in Section 3.2.

21.3.3 TAMP with Informed Metric Backtracking

Consider a search node (π, ω, s) of F-TAMP in which the metric feasibility test fails (Line 2). Here, action a is valid with respect to s but not to ω: its preconditions on symbolic variables are met, but those on metric variables are not. For example a take action with no free path to a grasp configuration.

In such a case, an informed metric backtracking procedure would list the motion, placement and grasping constraints responsible for the failure of a. It would find colliding obstacles for a free path to the target configuration, or for a stable pose on the target support. It would seek previous motions or manipulations which constrain a. These violated constraints are then checked with respect to the samplings previously performed in the current π.

Actions in π are analyzed in the reverse order seeking *culprit* actions, i.e., those which sample variables involved in the violated constraints. At such a culprit action a_i, alternative samples are checked such as to satisfy the violated constraints without breaking the remaining part of the plan $\langle a_{i+1}, \ldots, a \rangle$. If changes in a_i that satisfy a affect an action a_k in between the two, alternative samples for a_k might be analyzed similarly. This is an instance of dependency-directed backtracking focused on the sampling of numerical variables.

To be efficient, an informed backtracking procedure requires a good representation and processing of the following metric constraints:

- *Motion constraints* and collisions restraining movements. When no free path is found for an action, the motion planner has to provide a minimal set of removable obstacles whose removal can open a free path (see the Obstacle-to-move function of Example 21.10).
- *Placement constraints*: correspond to the set of stable poses in Q^o_{sta} on available supports or locations.
- *Grasping and ungrasping constraints*: correspond to the set of pairs $(\boldsymbol{\theta}, \mathbf{q})$ in Q^o_{grasp} taking into account geometric and kinematic constraints. One needs to know which grasp $(\boldsymbol{\theta}, \mathbf{q})$ is reversible for a desired ungrasp pose $\mathbf{q}'$, i.e., giving a transformation $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$ such that $\boldsymbol{\theta}' = \mathcal{H}^{-1}_{\boldsymbol{\theta},\mathbf{q}}(\mathbf{q}')$ exists.

An informed backtracking procedure can benefit from approximations of the metric constraints allowing the synthesis of a constraint graph over all the samples in responsible actions of violated constraints. Two such approximations can be considered:

- Discretization of the space on available supports for poses into a finite set of placements (possibly with a hierarchy of boxes); discretization of possible grasps over a finite set of alignements of the axes of the end effector and the object and their relative positions.
- Linearization of the placement and grasping constraints.

These approaches allow globally addressing the violated constraints in a metric backtracking point such as to seek a consistent assignment over all sampled variables in responsible actions. They can use respectively CSP techniques and linear programming methods, and can benefit from preprocessing stages, e.g., about supports and grasps. Both approaches guarantee correctness but usually not completeness: there may exist a consistent assignment that cannot be found through discretization or linearization.

Informed metric heuristics. Metric constraints is also needed for the definition of good value functions and heuristics to guide choices about which movable objects to move out of the way and where to put them; which object pose to sample for a placement on some support that is feasible and less clustering for remaining actions of the plan; which feasible grasp would lead to the largest set of possible ungrasps. Here too, approximations by discretization or linearization, and preprocessing are needed. Their integration into a prior learning stage can be very beneficial (see Section 22.2).

21.3.4 TAMP with a Hierarchical Refinement Planner

Let us first remark that what we just saw with F-TAMP can be adapted to an HTN schema, such as TO-HTN-Forward. The notions of metric backtracking and heuristics can be taken into account in the specification of HTN methods. Example 21.10 illustrates this approach, that will not be further developed here.

In this section, we adapt instead to TAMP the hierarchical refinement approach seen in Part V. RAE/UPOM were partly motivated by the use of operational models to efficiently interleave acting and planning, while doing enough lookahead to ensure a reliable behavior. This motivation holds for interleaving motion and task planning.

Methods in RAE can call any programme, e.g., the geometric reasoner mentioned earlier for handling functionally defined changes. We introduce in refinement methods a syntactic construct denoted "sample" which explicits sampling choices in metric ranges, e.g., in C_{free}, Q^o_{grasp} or Q^o_{sta} to move, grasp or pose an object. For UPOM, sampling marks a possible branching points in Monte Carlo rollouts adapted to the metric state variables.

As in action schema, a condition marked "sample" in a method corresponds to a set of choices. When this set is empty, the method is not applicable. Otherwise, alternative choices in the set can be tried. The method fails if all chosen samples lead to failure. Possibly, incremental sampling may be performed with smaller error margin, as seen in Section 21.1. Note that the set of choices for metric variables is defined functionally from metric properties.

Example 21.10. The DWR domain in Example 14.1 structures a harbor space as a graph of discrete Locations. Let us extend this representation with geometric models of each location in local reference frames. We assume the harbor to be a connexe graph; roads between adjacent locations are always traversable by robots; each location has a "gate", i.e., a free configuration allowing entry in and exit from that location.

The robots in this domain have the following primitive actions:

- traverse(r, l, l'): robot r traverses from location l to adjacent l';
- move$(r, \boldsymbol{\theta}, \boldsymbol{\theta}', \wp)$: r follows a free path $\wp$ from configuration $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$;
- grasp$(r, o, \boldsymbol{\theta}, \mathbf{q})$: r in configuration $\boldsymbol{\theta}$ grasps container o in its pose $\mathbf{q}$;
- ungrasp$(r, o, \boldsymbol{\theta}, \mathbf{q})$: r in $\boldsymbol{\theta}$ ungrasps o from its grasping pose $\mathbf{q}$.

Grasping and ungrasping may be performed only for a container on a reachable pose $\mathbf{q} \in Q^o_{grasp}$. Robots can perform the following tasks:

- Transport(r, o, l): r transports container o from its current pose and location to some stable pose in location l;

- Navigate(r, l, l'): r navigates from its current configuration in location l to the gate of l';
- Follow(r, way): r traverses the roads between the adjacent locations in the list *way* until the gate of the last one.
- Take(r, o): within its location, r takes o from its current pose and carries it to the gate of that location;
- Put(r, o): within its location, r carries o from the gate of that location to a stable pose, it then moves to a rest configuration;
- Remove$(r, o, \wp)$: r removes an obstacle o to a pose out-of-the-way to make $\wp \in C_{free}$;
- Remove-list$(r, lobs, \wp)$: r remove obstacles in the ordered list *lobst*.

Note that Transport fixes a destination location, but leaves the destination stable pose unspecified, to be chosen in the Put task with some heuristics.

The methods for specifying the tasks use the functions Grasping and Ungrasping. The handle the removal of obstacles, we modify the output of these two procedures such as to return: (*i*) a pair $(\boldsymbol{\theta}', \wp_{[\boldsymbol{\theta},\boldsymbol{\theta}'})$ if there is a feasible grasp/ungrasp configuration $\boldsymbol{\theta}'$ and a free path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ to reach it, (*ii*) a pair $(\boldsymbol{\theta}', \varnothing)$ if there is a feasible grasp/ungrasp $\boldsymbol{\theta}'$ but no free path, and (*iii*) nil otherwise. In addition, we rely on the following functions:

- Route(l_0, l_n): returns a sequence $\langle l_1, l_n \rangle$ of locations, with l_{i-1} adjacent to $l_i, 1 \leq i \leq n$; it implements a graph search which always succeed since the graph is connexe.
- Gate(l): returns a configuration at the gate of l.
- FreePath$(r, \boldsymbol{\theta}, \boldsymbol{\theta}')$: returns $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']} \in C_{free}$ for r if such a path exists, or nil otherwise; it uses a motion planner, e.g., Incremental-RRT.
- Obstacles$(r, \boldsymbol{\theta}, \boldsymbol{\theta}')$: returns a path $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ and an ordered list of removable obstacles whose removal would make $\wp_{[\boldsymbol{\theta},\boldsymbol{\theta}']} \in C_{free}$.
- Out-of-the-way$(r, o, \boldsymbol{\theta}, \mathbf{q}, \wp)$: returns a pair $(\boldsymbol{\theta}', \wp')$ of a configuration $\boldsymbol{\theta}'$ where r may ungrasp o in a stable pose that is not an obstacle to the path $\wp$, and a free path $\wp'$ to $\boldsymbol{\theta}'$. It relies on Ungrasping.

Obstacles relies on the procedure NAMO: it searches for a free path $\wp'_{[\boldsymbol{\theta},\boldsymbol{\theta}']}$ in a configuration space *without* movable obstacles and identify the list obstacles to remove. This list is ordered such that the first obstacle is reachable from $\boldsymbol{\theta}$; its removal would make the second obstacle reachable and removable, etc. Out-of-the-way looks for poses in current location where to put an obstacle to be removed such that it no longer obstructs $\wp$. We assume enough space in each location to always leave “out-of-the-way” poses where to put the containers to be removed. However, some of these poses may interfere with the follow up of the task; a good choice of an out-of-the way pose is needed.

A method for the simple navigation task from l to l' seeks by sampling a free path to the gate of l then a route in the graph between l and l':

```
m1-navigate$(r, l, l')$
      task: Navigate$(r, l, l')$
       pre: loc$(r) = l, l \neq l'$, config$(r) = \boldsymbol{\theta}$, Gate$(l) = \boldsymbol{\theta}'$
    sample: $(\wp \dot{\leftarrow}$FreePath$(\boldsymbol{\theta}, \boldsymbol{\theta}')) \neq$ nil
      body: move$(r, \boldsymbol{\theta}, \boldsymbol{\theta}', \wp)$
            $way \leftarrow$ Route$(l, l')$
            Follow$(r, l, way)$
```

```
m1-follow$(r, l, way)$
    task: Follow$(r, l, way)$
     pre: $way \neq$ nil, loc$(r) = l$
    body: $l' \leftarrow$ pop$(way)$
          traverse$(r, l, l')$
          Follow$(r, l', way)$
```

If there is no free path to the gate of location l another Navigate method would move out obstacles (see Exercise 21.4). Termination methods for these two tasks check respectively when $l = l'$ or $way =$ nil and simply return success.

The Transport task requires to navigate to where the container o is, take o, navigate to destination, then deliver o:

```
m1-transport$(r, o, l)$
    task: Transport$(r, o, l)$
     pre: cargo$(r) =$ nil
    body: Navigate$(r$, loc$(r)$, place$(o))$
          Take$(r, o)$
          Navigate$(r$, place$(o), l)$
          Put$(r, o)$
```

Another method for this task when cargo$(r) \neq$ nil would use the task Put to place cargo(r) out-of-way in the same loc(r).

The method for m1-take move to and grasps the target container if there is a free path to a grasp configuration. If there is a feasible grasping configuration but no free path to it, m2-take uses Obstacles to get a path to $\boldsymbol{\theta}'$ and list of obstacles whose removal would make this path free.

```
m1-take$(r, o, \boldsymbol{\theta}, \mathbf{q})$
      task: Take$(r, o)$
       pre: loc$(r) =$ place$(o)$, config$(r) = \boldsymbol{\theta}$, place$(o) = \mathbf{q}$
    sample: $((\boldsymbol{\theta}', \wp) \dot{\leftarrow}$ Grasping$(r, o, \boldsymbol{\theta}, \mathbf{q})) \neq$ nil, $\wp \neq \varnothing$,
      body: move$(r, \boldsymbol{\theta}, \boldsymbol{\theta}', \wp)$
            grasp$(r, o, \boldsymbol{\theta}')$
```

```
m2-take($r, o, \boldsymbol{\theta}, \mathbf{q}$)
        task: Take($r, o$)
         pre: loc($r$) = place($o$), config($r$) = $\boldsymbol{\theta}$, place($o$) = $\mathbf{q}$
      sample: (($\boldsymbol{\theta}', \wp$)$\overset{\cdot}{\leftarrow}$ Grasping($r, o, \boldsymbol{\theta}, \mathbf{q}$)) $\neq$ nil, $\wp = \varnothing$,
        body: ($path, lobs$) $\leftarrow$ Obstacles($r, \boldsymbol{\theta}, \boldsymbol{\theta}'$)
              Remove-list($r, lobs, path$)
              Take($r, o$)
```

Method m1-remove-list removes from *path* the obstacles in the ordered list *lobs*. A termination method would stop when $lobs = \langle\rangle$. Method m1-remove seeks a *path1* to a configuration to grasp the obstacle to remove, then a *path2* for a configuration to remove it out-of-the-way from the initial *path*.

```
m1-remove-list($r, lobs, path$)
      task: Remove-list($r, lobs, path$)
       pre: $lobs \neq \langle\rangle$
      body: $o \leftarrow \text{pop}(lobs)$
            Remove($r, o, \wp$)
            Remove-list($r, lobs, path$)

m1-remove($r, o, \wp$)
        task: Remove($r, o, \wp$)
         pre: config($r$) = $\boldsymbol{\theta}$, $pose(o) = \mathbf{q}$
      sample: (($\boldsymbol{\theta}_1, \wp_1$)$\overset{\cdot}{\leftarrow}$Grasping($r, o, \boldsymbol{\theta}, \mathbf{q}$)) $\neq$ nil, $\wp_1 \neq \varnothing$
              (($\boldsymbol{\theta}_2, \wp_2$)$\overset{\cdot}{\leftarrow}$Out-of-the-way($r, o, \boldsymbol{\theta}, \mathbf{q}, \wp$) $\neq$ nil, $\wp_2 \neq \varnothing$
        body: move($r, \boldsymbol{\theta}, \boldsymbol{\theta}_1, \wp_1$)
              grasp($r, o, \boldsymbol{\theta}_1, \mathbf{q}$)
              move($r, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \wp_2$)
              ungrasp($r, o, \boldsymbol{\theta}_2$)
```

A complete specification for this example would need additional methods (see Exercise 21.4 to 21.11). □

This example illustrates how hierarchical refinement methods can be used to interleave motion and manipulation with tasks in the planning and acting approach of RAE/UPOM. The methods require specific functions for sampling in $C_{free}, Q^o_{grasp}\ Q^o_{sta}$ (e.g., FreePath, Grasping or Ungrasping). For the sake of readability, Example 21.10 has presented the methods in a modular way. However efficiency considerations for the computationally demanding motion planning functions should lead to merge some methods to avoid querying for a same path more than once (e.g., in m1-take and m2-take) into a method with a more complex body field.

Since RAE can execute methods with any code and since UPOM uses simulation of tasks with the code in methods we already have the building blocks needed to use RAE/UPOM on TAMP problems. However, a few modifications in the pseudo-code of Chapter 14 and Chapter 15 are required:

- Recall the motion planning issue of probabilistic completeness and proximity to obstacles, which led to the Incremental-MP procedures. These algorithms handle a failure to find a path with additional sampling and narrower error margin. Here, the conditions marked "sample" indicate that, on failure, further sampling may permit to find a solution. This needs to be taken into account in the procedure Progress: a failure of a method may not mean that all the same instances of that method fail in this state. Before retrying another method instance (in line 2 of Progress), it is desirable to check if the failure can be dealt with through further sampling.

- These considerations hold also for UPOM. More precisely, we considered in UPOM three cases for handling the current step of the method under evaluation: an assignment step, a primitive action step, and a task refinement step (respectively in lines 2, 3, and 5). We need to process the "sample" conditions of methods explicitly. A simple way is to do it as for handling a primitive action step, i.e., progressing recursively on the current rollout with a sampled value if one is found, or returning failure for that rollout if no sample exists. An elaborate processing of "sample" would take into account the number of sampled values in $N_{\text{stack},s}(m)$, and hence in $Q_{\text{stack},s}(m)$ (which approximates the utility function value).

Note that learning with CORL can provide a domain-specific method-value function Q_0 informative for an efficient sampling strategy. This issue will be further develop in the next chapter.

21.4 Discussion and Bibliographic Notes

The problems of motion and manipulation planning in robotics are very rich and draw numerous contributions, a full tribute to which is beyond the space limitations of this section. We focus the following discussion mainly on the recent work on TAMP problems, after a brief review of motion planning and manipulation-planning literature.

21.4.1 Motion Planning

Motion is a fundamental function for autonomous robots, thoroughly studied. Motion planning benefits from several textbooks, among which those of Latombe [684], Choset et al. [231], and LaValle [685]. The topic is also covered in most recent robotics textbooks, e.g., [743], and the handbook of robotics [1014]. The following is a brief summary of a few main developments.

The basic "piano mover problem" for planning the motion of a free body in a constrained space was characterized as PSPACE-hard [940]. The configuration space representation for motion planning is due to [740]. Early complet planners (e.g., [993, 199]) where too inefficient. Practical approximations have been quickly proposed, e.g., with space decomposition [180], potential fields [603], or their combination [225]. Probabilistic sampling and PRM methods have been introduced in [593], and proved to be probabilistically complete [84]. They are now the dominant and widely used approach in motion planning.

The visibility-based vPRM algorithm appears in [1022]. RRT techniques are due to [648]. Asymptotically optimal algorithms such as PRM*, RRT* and bi-directional RRT* have been studied in [584, 560].

Some motion planners can cope with dynamically changing environments, e.g., [547]; they are surveyed in [803]. Motion planning for autonomous driving was a benchmark in the field with early demonstrations in bounded area, e.g., for car parking. Open road motion planning raises additional issues and needs to be interfaced hierarchically with route planning; several approaches are reviewed in [863, 242, 1090].

The presented PRM and RRT family of algorithms do not handle the velocity, acceleration, force and torque dynamic constraints of robots. These are addressed by *kinodynamic motion planners*, e.g., in [302, 686, 896], and [1130] with feedback control algorithms.

In applications such as moving a glass without spilling its content, the movement to be planned is constrained. This is referred to as *motion planning with constraints*. The basic techniques combine constraint satisfaction techniques with sampling in the configuration space [607, 272].

Other techniques seek to obtain plans that are robust to the uncertainty of the models and to the sensory-motor noise in the robot localization and motion control, and to account for the possibly growing localization uncertainty with movements aware of landmarks [487].

Finally, let us mention the numerous links between motion planning and navigation problems (see Section 20.3). When a map of roads is given, as in autonomous driving, the problem is decomposed into finding a *way* (as in Example 21.10) to be followed with reactive control, e.g., elastic bands, then planning precise movements when close to the target or in off-road areas, e.g., a parking [348]. When a mapped is to be learned, as with SLAM, additional techniques are needed and surveyed in [1185].

21.4.2 Manipulation Planning

The strong demand for automated manipulation-planning capabilities motivates numerous investigations in this area, surveyed in, e.g., [1066] Historically, early manipulation planners discretize the manipulation space and synthesize a manipulation graph which is searched for a solution, e.g., [22].

The problem in Example 21.7 and other similar complex instances of manipulation planning is addressed in [1023] with a multiple query probabilistic roadmap approach. The algorithm is similar to PRM, but instead of sampling points in C_{free}, it samples pairs $(\boldsymbol{\theta}, \mathbf{q})$ in a space similar to $\mathcal{M}^o$. It considers the couple $(\mathfrak{R}, o)$ and the stable supports as a *closed chain* of connected kinematic links. Single-query approaches are however conceptually simpler.

The previous formulation refers to problems with completely specified target configurations. In practice, a task only constrains feasible configurations. For example, a grasp constrains the end effector configuration in $\boldsymbol{\theta}_g$. It is possible to decompose the problem by finding a movement of the base of the robot to an accessible configuration for the object, then plan a movement of the arm to a grasp position. However, the manipulation of an object can require intermediate poses at different moments, or the manipulation of other interfering objects. It is necessary to change the structure of the search space according to the grasps and poses of objects handled [1023]. In addition, the above decomposition is not always feasible. For example, a humanoid robot requires a coordinated movement of all its *dof* for its body and all limbs [582] (Figure 20.1). Further, sensing and visibility issues bring additional constraints, e.g., planning a motion that avoids occultation between a camera carried by the robot's head and its hand, or allowing for visual servoing [220].

21.4.3 Combined Task and Motion Planning

Section 21.3 details two possible approaches to TAMP: forward search and hierarchical refinements. Several instances of the former have been studied in the literature, the latter is original. These two approaches gives a very partial view of the area of TAMP methods, which recently mushroomed into hundreds of quite creative contributions. Since TAMP is exemplary of heterogeneous symbolic–metric planning problems, we devote here a longer discussion for broader view of the main classes of TAMP methods.

TAMP with generalized roadmap approaches. It seems natural to address TAMP as a *multi-modal motion planning* problem, a modality being a configuration space (robot alone, in contact with or carrying an object, etc.). This is exemplified in [483], which relies on a *task graph* G. A node in G is labeled by an action a and associated with a configuration space F_a where a is achievable. An edge (a, a') denotes that a' is feasible after a; it corresponds to the intersection $F_a \cap F_{a'}$, called the *transition space*. Transitions are more constrained; they can focus sampling. The procedure builds G incrementally, it computes a PRM of F_a for each new node. It connectes the PRMs through their transitions into a path from a start node to a goal node. G is heuristically searched with a focus on sampling transitions, allowing to prune unsuccessful paths. The approach has been illustrated with legged robots.

The method called *Probabilistic Tree of Roadmaps* (PTR) [482] illustrates a generalization of RRT. Preconditions and effects in action models use domain-specific procedures requiring fully specified states to be evaluated. PTR extends the search tree in a forward manner by sampling k successors for every node. As RRT, only one such successor is added to the tree, randomly chosen such as to keep an asymptotically uniform distribution of states. The algorithm is probabilistically complete, but it does not use the symbolic state variables and relations to constrain or focus the tree extension.

DARRT [483] is another RRT-like TAMP planner with a similar search space. It requires and can benefit from domain-specific heuristics about the effects and preconditions of each action. It has been illustrated with non-prehensile manipulation actions, e.g., push a plate to the border of a table, grasp its edge, pick it up, move it, put it down, then push it to a destination pose. It has also been extended with bidirectional and hierarchical search. The hierarchical extension works by first generating a free path for the object alone, annotated with the needed manipulation actions. Each action in that path is a subgoal to be further refined.

TAMP with state-space search. Several approaches try to leverage symbolic causal relations using a task planner to guide TAMP. Asymov [197] offers an early illustration. It combines the metric-FF task planner with motion planning. It defines a *place* as a state in the task planning space, as well as a roadmap in the corresponding configuration space. It expands roadmaps along with the search; it can reuse and amortize them over several places. This approach has been extended to multi-robot cooperation over complex manipulation and assembly tasks.

FFRob [386] is a different extension of FF with heuristic search in a symbolic-metric space. It builds incrementally a *Conditional reachability graph* (CRG), which encodes the connectivity of sampled configurations. An edge is a path between two connected configurations labeled with traversability conditions on the poses of objects what robot grasps. Action-specific sampling procedures need to be programmed. "Interesting" samples of poses and configurations are initially drawn to access to or put away movable objects, or to connect configurations. This set is progressively extended while planning. CRG permits to easily find if a node in the graph is reachable through sampled nodes from a search state. It is used for finding applicable actions in a state, and for computing extension of classical heuristics such as HFF. CRG is convenient for solving multiple motion-planning queries in similar environments. FFRob is probabilistically complete; its probability of failure decreases exponentially in the number of iterations.

The Hybrid Backward-Forward (HBF) algorithm [384] is a variant of FFRob which uses backward search to produce successors and distance estimates for the main forward search. HBF uses a structure called *constrained operating subspace* (COS) which is a roadmap sampler of partial states from preconditions to effects, as well as a transition and an intersection sampler for connectivity. COSs encapsulate all the knowledge about the domain and lead to lead to extended action models with the needed metric transitions (e.g., paths for a move). A relaxation heuristic gives good performance in object sorting and placement problems (e.g., plans of about 70 actions in a minute CPU).

The LD-CTAM approach [669] considers a forward search strategy that can be parametrized. Down to a depth d^* of the search tree, it combines symbolic and metric (by sampling) action instantiation. An instantiation may fail due to obstacles, kinematic reachability, or failure of the motion planner; in which case, metric backtracking occurs. If metric backtracking fails (after a cut-off number of trials), symbolic backtracking is triggered. Beyond d^*, the search ignores metric values at the task planning level until reaching a goal, then it evaluates and instantiates metrically this candidate plan. If needed, backtracking is performed at the metric level, then at the symbolic level. A simplified analysis, empirically tested, shows that there is an optimal depth d^*, which depends on a metric pruning factor.

Another option, illustrated in [354], compiles a TAMP problem into a classical planning one. At a costly preprocessing stage, the method discretizes the metric space and computes with a motion planner two finite graphs of possible feasible configurations and their transitions. These are used by a classical task planner.

Completeness of the planner and the method is ensured for the chosen discretization, which has to be refined if no solution is found.

Among many other contributions to the popular state-space search idea for TAMP, let us also mention a basic extension of PDDL schemas with *semantic attachements*, (i.e., the needed metric specific procedures) developed in [306], or an original weighted forward-search using LTL and finite automata for tasks planning together with RRT for motion planning [779].

TAMP with constraint satisfaction methods. Several contributions relies on various ways of leveraging metric constraints to bound, prune and focus the search of a TAMP planner. The approach of [670] uses a constraint-based method to prune the search. A sequence of actions synthesized by a task planner is mapped to a set of linear constraints which condition the existence of a metric instantiation of this sequence. Violated constraints allow safely pruning that sequence. But satisfied constraints do not guaranty a feasible metric instance of that sequence, since the used constraints rely on a coarse metric representation of symbolic states. Constraints do not take into account intermediate configurations between states, neither do they express the entire set of metric relations in each state. Further, part of these constraints rely on predefined finite sets of poses and grasps. With these simplifications, the constraints for manipulation tasks are generated at each iteration of a depth-first search; they are expressed as two sets of linear equality and inequality constraints, checked with an LP solver. Empirical assessment shows this constraint handling method pays off when compared to straightforward backtracking, in particular when the domain is highly constrained (e.g., filling a cup held by the left hand with a water bottle held by the right hand).

In [142], additional constraints are used for informed symbolic and metric backtracking. A state-space forward search (with totally ordered HTN) uses symbolic backtracking on actions, and metric backtracking on sampled metric values in the chosen actions (poses, grasps, paths). Three types of constraints about the movement kinematics, the object placements and their grasps are managed with linear programming and interval filtering algorithms. Heuristics about the number of detected collisions in object placements and the number of violated kinematic constraints in picking and placing objects are also used. A comprehensive experimental evaluation for complex tasks (dual-arm manipulation and warehouse management by a forklift robot) shows feasibility with a modest scaling up (up to two dozen actions).

A more elaborate metric failure analysis is developed in [668]. A failure has a *culprit*, i.e., a parsimonious explanation used for pruning the current sequence of actions, and any future sub-plan that have that culprit. Informally, a culprit, as in diagnosis and abduction reasoning, is a subset of hypotheses that gives a complete and parsimonious explanation of observations. Two culprit detection methods are proposed here. The first one extends the linear coarse metric constraints of [670] to account for the current sequence of states. The second method considers intermediate configurations between symbolic states and their causes of failure. The method computes for each objet a bounding box of all the poses that the object can possibly occupy at some step. The bounding boxes are represented as a network of linear constraints, which is used to find possible violations of kinematic constraints and collisions. The exploitation of the found culprits uses a strategy similar to that of CSP and SAT solvers: an inconsistency is expressed as a clause whose negation is added to the base of clauses. This motivates the use of an ASP task planner, since culprit trial can readily be integrated in ASP. The planner synthesizes a full symbolic plan for the problem at hand that avoids all known culprits. That plan is analyzed by the geometric reasoner, which either succeeds with a full metric instantiation of the plan or generates additional culprits. The geometric reasoner looks for inconsistent spatial relations between objects with the bounding box method, then for infeasible subsequences of actions with the linear coarse constraints method, and if none is found it seeks a full metric instantiation of the plan. If this last step fails, the geometric reasoner does not identify colliding obstacles that obstruct the path planner (a hard problem). It assumes the culprit to be the sequence of actions for which motion plans have been actually found: this sequence makes the remaining part of the plan not feasible; the task planner is required to avoid it in future trials. The approach is not complete because of the latter point and the bounding box method requirements, which are not a necessary condition. Empirical evaluation on complex manipulation problems demonstrates a reasonable scaling up (20 actions planned in about 2 minutes, most of it being in geometric reasoning).

A CSP solver on discretized metric space is used in [739]. A task planner gives a candidate abstract skeleton plan leaving unbound the metric variables in action preconditions and effects. The CSP solver determines grasps, object placements, robot configurations, and paths that satisfy the constraints. An elaborated pre-processing phase allows building the discretized free space and a spanning tree which represent the connectivity of that space. The collision, grasp and path constraints are augmented with a few necessary but not sufficient simple constraints (e.g., bounding boxes) to prune the CSP search. The approach is evaluated on object arrangement problems, with possible regrasps consistent with a destination pose. It outperforms standard backtracking, in particular when there is no solution.

The method of [272] uses a CSP solver (namely SMT) for task planning coupled to RRT for metric planning. The latter uses the popular kinematic tree model for the configuration spaces, available in several software packages, which efficiently handles the changing kinematic constraints as objects are grasped, moved, and released. A failure in metric planning entails additional constraints in the task CSP. Since task planning with CSP requires a *length* parameter (maximum number of actions in searched plans), the approach links this parameter to the maximum number of samples (or the timeout parameter) of RRT. Both are jointly increased an iterative deepening procedure. The constraint encoding is polynomial in the number of objects and locations in the domain. The approach benefits from generalized constraints from the analyzes of failures and detected collisions. It is probabilistically complete and scales up to about 100 objects in arrangement problems in about 100s, a tenth of which in task planning.

A meta-CSP approach is explored in [1058, 752]. It has meta-variables and meta-constraints related to lower level CSPs, four of which are considered about respectively time, space, resource and causal constraints. A planning problem is stated as a global constraint network that solves conflicts in the four types of CSPs. Resolvers for a conflict are additional actions, temporal, or spatial constraints. Both time and space are expressed in finite CSPs using a tractable subset of *interval algebra* and its 2D extension to the *rectangle algebra*. Numeric bounds on distances and time points can be handled. Each resolver for a conflict is checked for time and space consistency. The meta-variables range over lower constraints; the meta constraints correspond to consistent ways of combining lower level constraints. A solution is computed by a backtrack search algorithm at the meta level relying on the ground solvers, improved in CHIMP [752] with an HTN planner adapted to the meta-CSP framework. It uses constraints on the HTN task decomposition order, as well as lower and meta-constraints. The main advantage of the approach is to handle time in TAMP for problems with deadlines, with a drawback of a limited handling of space and movements. It has been tested on simple object-placement problems with plans with up to 40 actions.

Several other CSP-based approaches have been studied, such as [326] or [837], with respectively ASP and SMT solvers, using variant of the above principles and additional ideas.

TAMP with hierarchical approaches. Several contributions to TAMP rely on a hierarchical decomposition of the task and motion levels (as in Figure 21.7). The approach of [754, 755] uses *"angelic nondeterminism"*. The latter assumes that a lower level to which a choice has been deferred, will be able chose the best option (see Section 5.5.3). A task plan is sequence of on symbolic actions, each can be decomposed at the metric level in different ways. The possible decompositions of a task plan are the compatible decompositions of its actions. A plan is acceptable if it has at least one feasible decomposition. The lower level decomposes, when needed, each abstract action by choosing a feasible decomposition, if there is one. The key idea is to rely on a lower bound of the set of feasible decompositions of an task plan such as to make sure that this set is not empty. Lower bounds are computed by running simulations of action decompositions using random values of state variables. The planner relies on these estimates for searching in the abstract state space.

The HPN hierarchical approach [566] estimates the metric feasibility of its actions with "*Geometric Advisers*". Advisers do not solve motion planning problems. They provide heuristic informations about how metrically feasible is a given symbolic action. Task planning continues with the green light from advisers until reaching a complete plan. Task planning in HPN relies on a goal regression search hierarchized according to a user-defined partition of state variables into a few levels of criticality. The regression in the search space is pursued until the preconditions of the considered action (at some hierarchical level) are met by current world state. At acting time, each step that requires movements triggers a full motion planning. This approach relies on two assumptions: metric preconditions of abstract actions can be calculated quickly and efficiently by the geometric advisers; subgoals resulting from decomposition of action are executable in sequence. The approach is not complete; the refinement of an action may fail despite the advices. For problems without dead ends allowing for "backtracking" at the acting level at a reasonable cost, the approach is efficient and robust.

HPN has been extended over several contributions, notably in [567, 568]. The main extensions are about handling epistemic actions, reasoning on what the robot knows and can sense, and about the uncertainty on object poses and the robot localizations. After each observation (assumed with Gaussian noise), a state estimator performs a Bayesian update of a distribution of the robot estimates about the current state of the world. Specific function are proposed to compute uncertainty parameters through regression, e.g., to find the minimum confidence required in the location of an object at the previous step, in order to guarantee some required confidence in its location as a result of an action. Since the method interleaves acting and planning, it monitors how a plan progresses in order to eventually try another action in case of a failure without replanning, or to opportunistically skip actions if acting can move ahead in the current plan towards the goal. This quite comprehensive approach to TAMP with numerous features allowing for robust robot behaviors, requires significant expertise in the specification of action models.

Other hierarchical approaches have been explored, such as [278, 412] relying on variables shared between the task level (with HTN) and the metric level, with a specific geometric reasoner called GTP, allowing for feasibility test of metric instantiations and informed backtracking.

TAMP with mathematical programming. TAMP has been addressed in [1101] as a relational mathematical optimization program with respect to a cost function over the final state, e.g., maximize the height of a cup in a stable assembly construction using boards and various objects. The method addresses the problem as a first order logic extension of a non-linear constrained program over trajectories formulated as follow:

$$\min_{x,\pi} f(x,\pi) \text{ such that } \pi \models K, \quad g(x,\pi) \leq 0, \quad h(x,\pi) = 0$$

where π is a task plan, x is a sequence of continuous trajectories of the robot and objects, K is a logic specification of the domain, f is the objective function; it is assumed that π entails on the trajectories x equality and inequality constraints denoted by h and g respectively. f takes into account control variables (on $x, \dot{x}$, and $\ddot{x}$); h and g integrate dynamic and stability constraints. The global optimization program is broken down over three successive approximation steps: (i) optimize over all metric parameters in the final state of π, (ii) optimize over all intermediate states in π given the results of (i), and (iii) optimize over all trajectories (discretized in time) given (i) and (ii). The first step is very efficient for binding parameters of actions in π, including those in early actions. Because of the constraints in the final state, this step is of lower dimensionality than the sum of all metric parameters of actions in π. These three steps are integrated within a Monte Carlo Tree Search (MCTS) over feasible skeleton plans. Step (i) is performed for each rollout. The best plans are optimized with step (ii), and, if feasible, with step (iii). The approach is tested in simulation with a simplified 3 DOF robot arm. It works for plans with up to 50 actions in as many seconds. This formulation of TAMP as a relational mathematical program offers interesting features such as handling dynamics and stability of assembled construction. It also demonstrates step (i) as an efficient heuristics for other TAMP approaches.

This quite long discussion is certainly not exhaustive of the ideas and methods explored in TAMP. Complementary approaches are reviewed in [387].

21.5 Exercises

21.1. In Figure 21.8, the white areas are C_{free} and the gray areas are C_{obs}. Suppose we're trying to create a roadmap, starting with the two points X and Y in version 1 of the figure.

(a) What is the smallest number of additional points needed to create a roadmap? Draw the roadmap.
(b) In your roadmap, are either X or Y redundant? Explain.
(c) Repeat questions (a) and (b) using version 2 of the figure.

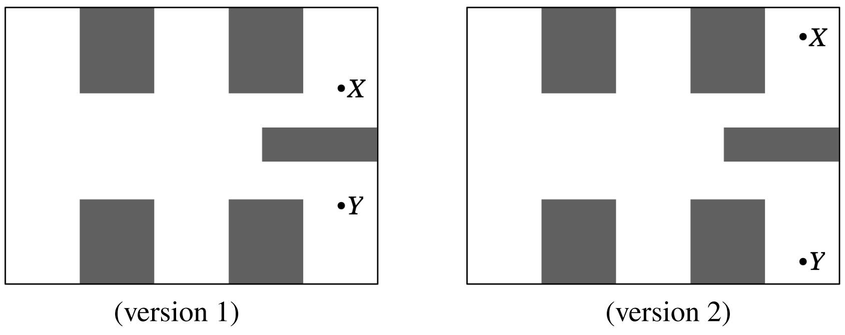


Figure 21.8. Two possible placements of points X and Y in a region with obstacles.

21.2. Write the pseudo-code for the interpolation procedure in p.487. Find a curve fitting procedure that returns the maximum distance to the set of points allowing to bound the distance to the points in the segments of in σ such as to simplify collision checking of $\wp_{[\theta_0,\theta_g]}$.

21.3. Consider a triangulation of C_{free} as in Figure 21.3(b), and let P be the set of triangle centers. Is it possible to define a roadmap of C_{free} based solely on the set P?

21.4. Write methods for the task Navigate of Example 21.10 to handle the case where there is no free path to a gate.

21.5. Write methods for the task Transport of Example 21.10 to handle the case where cargo$(r) \neq$ nil.

21.6. Write methods for the task Take of Example 21.10 to handle the case where Grasp-config(r, o) = nil.

21.7. Write methods for the task Take of Example 21.10 to handle the case where Grasp-config(r, o) = nil.

21.8. Write methods for the task Take of Example 21.10 to handle the case where Stacked-on$(o) \neq$ nil.

21.9. Write methods for the task Remove of Example 21.10 to handle the cases when FreePath$(r, \boldsymbol{\theta}_0, \boldsymbol{\theta}_1)$ = nil or when FreePath$(r, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$ = nil.

21.10. Write methods for the task Unstack of Example 21.10.

21.11. Write methods for the task Put of Example 21.10.

21.12. Define an action navigate(r, l, l') for Example 21.10 to go from the current robot location l and configuration to some free configuration, found by sampling, in a target location l'.

22 Learning for Movement Actions

In this part of the book, we discussed so far how movement and manipulation actions can be modeled, controlled, and planned for. The reader has noticed how complex it is to specify and develop the corresponding models. Modeling, controlling and planning with movement and manipulation actions is challenging. The high dimensional sensory-motor space and the needed integration of metric and symbolic state variables augment the challenges.

Machine learning addresses these challenges at the acting level as well as at the planning level. But ML in robotics faces specific problems:

- It does not benefit from the massive text and image data available over the web in other supervised learning applications.
- Experiments needed for RL are scarce, very expensive (compared to board games and computer games), and difficult to reproduce.
- Realistic sensory-motor simulators remain computationally costly.
- Expert human input for RL is often needed for, e.g., specifying or shaping reward functions or giving demonstrations and advices, but this expertise is scarce and costly; it needs to be used with parsimony.
- The functions learned are often narrow. Generalization of a learned behaviors and models across environments and tasks is challenging.

In the two following sections, we consider approaches for addressing some of these problems at successively the acting level, referred to as skill learning, then at the planning level. Recent progress of this fast moving field are discussed in Section 22.3.

22.1 Learning Sensory-Motor Skills

22.1.1 Robotics Skills

A *skill* refers to a physical elementary task that may require several movements and actions, e.g., to park a car along a sidewalk in a narrow spot. A complex task such as changing a flat tire may call for several skills. An industrial robot platform is generally endowed with a library of skills, e.g., for welding or bolting; similarly for a service robot. Often, the robotics literature illustrates skill development methods on physical games and sports such as archery, table tennis, or soccer, as in the popular RoboCup competitions.

A skill is defined with an operational model specifying how to combine a set of sensory-motor controls in order to perform the intended task. It uses metric representations for the state space as well as the action space, both of which are usually high-dimensional. Skill modeling and programming are complex undertakings, motivating the need for efficient skill learning.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

Skill learning methods are naturally set in the parametric RL framework. They seek to acquire from trial and error a policy that optimizes an objective function combining success reward and low cost. The basic approaches are those seen in RL, particularly the policy-based methods of Section 10.6.

Recall that the value-based Deep Q-learning can cope with high-dimensional state spaces, but is limited to discrete action spaces, while skills often involve continuous control. A possible option is to discretize the action space, but this is often problematic. For a robot of n *dof* with k discrete values in each *dof*, the action space is in $O(k^n)$, barely feasible for robots as Justin (Figure 20.6(a)), where $n \approx 50$. Hence, a policy-based RL method such as Policy Gradient, which can handle a high dimensional continuous action space, is appealing. Its use for learning robotics skills required several adaptations exemplified in the algorithm presented next.

22.1.2 Skill Learning

Let us study here Deep Skill Learning, an algorithm that combines the principles of Policy Gradient with insights from Deep Q-learning (relying on sections 10.5.1 and 10.6.2). This is an *actor/critic* algorithm which:

- decouples the evaluation of the target values from the ongoing updated networks to avoid instabilities due to targets close to current estimates,
- updates the networks over mini-batches randomly sampled from a replay-memory to take into account correlated successive observations.

Deep Skill Learning maintains and updates two main nets: $[\boldsymbol{\pi}_\theta]$ and $[Q_\omega]$. The actor net $[\boldsymbol{\pi}_\theta]$ takes a vector $s \in \mathbb{R}^n$ as input and gives as output a weight a vector over $a \in \mathbb{R}^m$. The critic net $[Q_\omega]$ takes the two vectors s and a as input and gives as output a scalar $Q(s,a)$. We also need to target nets, denoted $[Q_{\omega^-}]$ and $[\boldsymbol{\pi}_{\theta^-}]$, to decouple targets from values.

The algorithm runs over a number of episodes, each involving a finite number of steps until the termination of the task. It keeps a replay-memory $\mathcal{R}_M$ as a FIFO list recording the last N steps, where a new tuple remplaces the oldest one. $\mathcal{R}_M$ covers successive episodes, possibly several if N is large. A mini-batch $\mathcal{B}$ of k tuples is sampled uniformly from the replay-memory $\mathcal{R}_M$. The parameters $\boldsymbol{\theta}$ and $\boldsymbol{\omega}$ of the actor and critic nets are updated with Backpropagation algorithm, with an error term for ω and a gradient term for θ averaged out (**forall** loop in line 2) over the k tuples in the mini-batch $\mathcal{B}$. Note that the vectors $\boldsymbol{\delta}_\omega$ and $\nabla_\omega Q_\omega$ are of the same dimension as $\boldsymbol{\omega}$ (line 4). If m is the dimension of the action space and p is the dimension of $\boldsymbol{\theta}$, then $\nabla_\theta \boldsymbol{\pi}_\theta$ is a vector of the dimension p, the Jacobian matrix $\nabla_a Q_\omega(s, \boldsymbol{\pi}_\theta(s))$ is of dimension (p,m) giving for the product a vector $\boldsymbol{\delta}_\theta$ of dimension p (line 5).

The target y is defined as in Policy Gradient; it depends on Q_ω as well as $\boldsymbol{\pi}_\theta$. Hence we need two additional networks to decouple the target from current action-value. These two nets $[Q_{\omega^-}]$ and $[\boldsymbol{\pi}_{\theta^-}]$ are used to compute the target $y = r(s,a,s') + Q_{\omega^-}(s', \boldsymbol{\pi}_{\theta^-}(s'))$. Their parameters are not updated as for $[Q_\omega]$ and $[\boldsymbol{\pi}_\theta]$. They track with a delay those of the main nets with a linear combination of old and new parameter values: $\boldsymbol{\theta}^- \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}^-$, and $\boldsymbol{\omega}^- \leftarrow \tau\boldsymbol{\omega} + (1-\tau)\boldsymbol{\omega}^-$, for $\tau << 1$.

```
Deep Skill Learning
    initialize critic net [$Q_\omega$] and actor net [$\boldsymbol{\pi}_\theta$]
    initialize the replay memory $\mathcal{R}_M$
    $\theta^- \leftarrow \theta$ ; $\omega^- \leftarrow \omega$                                   // initialize target nets
    for each episode do
        randomly draw a starting state $s$ from $S_0$
        until episode termination do
1           $a \leftarrow$ Select($s$)                                   // selects $a \in Applicable(s)$
            perform action $a$
            observe resulting state $s'$ and reward $r(s,a,s')$
            push($(s,a,s',r(s,a,s')), \mathcal{R}_M$)                          // FIFO replay memory
            $\mathcal{B} \leftarrow$ set of $k$ tuples uniformly sampled from $\mathcal{R}_M$
            $\boldsymbol{\delta}_\omega \leftarrow [0,\dots,0]$ ; $\boldsymbol{\delta}_\theta \leftarrow [0,\dots,0]$
2           forall tuples $(s,a,s',r(s,a,s')) \in \mathcal{B}$ do
3               $y \leftarrow r(s,a,s') + Q_{\omega^-}(s', \boldsymbol{\pi}_{\theta^-}(s'))$
4               $\boldsymbol{\delta}_\omega \leftarrow \boldsymbol{\delta}_\omega + 1/k[y - Q_\omega(s,a)]\nabla_\omega Q_\omega(s,a)$
5               $\boldsymbol{\delta}_\theta \leftarrow \boldsymbol{\delta}_\theta + 1/k[\nabla_\theta \boldsymbol{\pi}_\theta(s) \times \nabla_a Q_\omega(s, \boldsymbol{\pi}_\theta(s))]$
6           $\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \alpha_\omega \delta_\omega$                                  // update critic network
7           $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \delta_\theta$                                  // update actor network
            $s \leftarrow s'$
8           $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$                           // update target nets
9           $\omega^- \leftarrow \tau\omega + (1-\tau)\omega^-$                           // update target nets
```

Algorithm 22.1. Deep Skill Learning with a deterministic policy gradient algorithm.

Essential choices are needed for instantiating Deep Skill Learning in an application. These are mainly related to the neural nets architecture and their nonlinear functions. A visual input would typically require convolution nets, while low-dimensional state and action spaces (e.g., a robot joint angles, velocities, and torques) may be handled with feedforward nets. In addition, several hyper-parameters require tuning. These are in particular the learning rates α_ω and α_θ, the delay τ for the parameters of the two target networks, the size of the replay memory $\mathcal{R}_M$ and the size k of the mini-batch $\mathcal{B}$. The Select function driving the exploration policy has to be chosen; for a continuous action space, it usually uses $\boldsymbol{\pi}_\theta$ plus some random noise.

The sampling of the mini-batch $\mathcal{B}$ from $\mathcal{R}_M$ may follow a more elaborate strategy than uniformly random, e.g., prioritize the last observations or use explicite priority estimates, as in the *prioritized sweeping* method. When the state and action components are heterogeneous, ranging over different scales, there is a need of uniform scaling methods, such as the *batch normalization*. This technique normalizes the inputs across the samples in a minibatch to have unit mean and variance; it maintains running averages of the means and variances for this normalization. It can be applied to hidden layers as well.

Deep Skill Learning benefits from the formal convergence properties of the deterministic Policy Gradient approach and the practical stability properties of Deep Q-learning. As a policy-based RL, it can take into account prior knowledge in the initial π_θ, which is a significant advantage to speed up learning. Instances and variants of this algorithm have been successfully tested in simulation and in physical experiments in a number of skills, ranging from simple ones, such as balancing an upright pole attached to a cart, to more elaborate pick-and-place of objects with a 7 *dof* arm (as in Figure 20.4), hitting a ball to a target using an arm equipped with e.g., a hockey stick, or bipedal and quadruped walking on a plane.

22.2 Learning for Task and Motion Planning

A TAMP planner explores joint symbolic and metric search spaces, the latter through sampling. It devotes significant efforts, with possibly additional sampling, to find out if a movement or a grasp are feasible (see Section 21.3). Recall the probabilistic completeness of motion and manipulation planning: the tighter a motion is, the longer it takes to compute the corresponding path; infeasible moves are the worst. Altogether, the metric part is the main computational bottleneck in TAMP.

The idea here is avoid as much as possible calling the planner for unlikely feasible movements, and to use learning techniques for recognizing those in advance. For that, we may rely again on the *"planning to learn"* paradigm (Figure 1.2): motion and manipulation planners are used offline to find feasible movements and grasps across a spectrum of tasks and contexts for the domain at hand. This prior training data is generalized into useful knowledge for online task and motion planning.

The approach is different from "*end-to-end learning*", used in the skill learning section, in which the actor performs reactively the learned skills. In TAMP, end-to-end learning is suitable for and has been deployed successfully in single manipulation tasks with fairly invariable environments. Variety in tasks and environment requires the flexibility of planning. In the "*planning to learn*" approach, planning gives training data to learning. The learner generalizes training into useful knowledge in order to better plan, with a different planner, before acting. While acting, additional experiences can feed in the training and learning base in a focused way.

The learning approach discussed here relies on the algorithms and methods seen for reinforcement learning (Chapter 10). This approach is in many ways akin to the one described for learning to guide RAE and UPOM (Section 16.1). In the latter, we learned a method-value function $Q_0(s, m)$ to be used by Guide and UPOM. Here, we learn an action-value function $Q_0(\omega, s, a)$ to be used by F-TAMP. The main novelty is that the learner takes as input *visual scenes* of the movement and manipulation problems to be learned, and as output target the success or failure of tried movements in these scenes. In previous approaches we addressed *learning in the feature space* (i.e., a coding of (s, a) or (s, τ)), while here we discuss *learning in the sensor space*.

Example 22.1. Consider a robot arm with a simple gripper (e.g., the Franka Emika of Figure 20.4) fixed on a table in an environment with a few horizontal support surfaces (shelves, trails, etc.) and a few movable objects of regular polyhedral shapes, as depicted in Figure 22.1[19].

Tasks in this environment are for example to fetch, sort or arrange in some order the movable objects. They can be planned for using F-TAMP and the take and place action schemas of Example 21.9. □

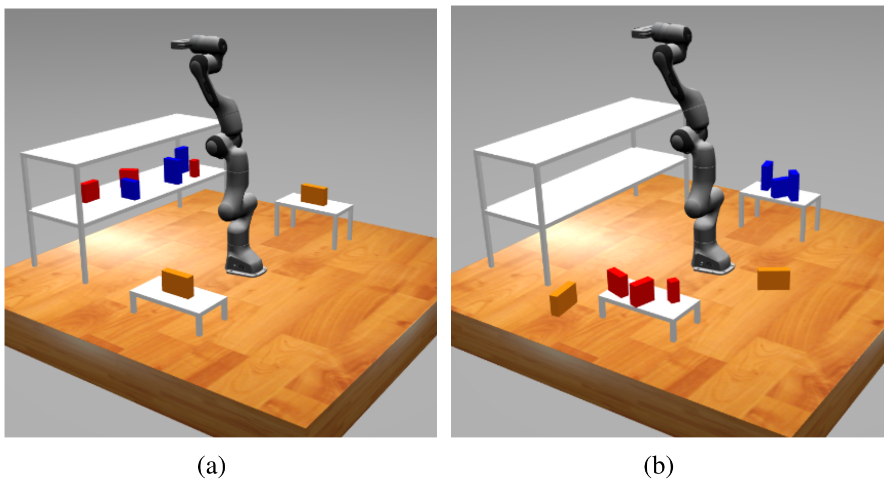


Figure 22.1. A simple TAMP problem where the task is to sort the eight movable objects according to their color from their initial poses in (a) to any poses on the support surfaces shown in (b) (figure from [19]).

Recall that F-TAMP depends strongly on how effective is the action-value function Q_0 for focusing the search. This function is defined as the ratio $r_s(\omega, s, a)/h(\omega, s, a, g)$, where $r_s(\omega, s, a) \in [0, 1]$ is the expected success reward of action a, and $h(\omega, s, a, g) \in \mathbb{R}^+$ is the remaining distance to the goal after a (see Equation 21.1). We can use classical planning heuristics on the symbolic part of a TAMP domain to define h.

We focus here on how to learn the success reward function $r_s(\omega, s, a)$ for the actions of a class of TAMP domains. To be concrete, let us illustrate the approach on the class of fixed robot manipulators with parallel grippers surrounded by a set of support places dealing with polyhedral objects in arrangement and sorting tasks, as illustrated in Figure 22.1. A domain Σ of this class uses the take and place actions of Example 21.9. Since the learning procedure is very much similar to that of Section 16.1, we briefly outline here its main steps, insisting on its specifics.

Training set generation. We randomly draw a set of TAMP problems in Σ where the goal differs from the initial state by the displacement of a *single* object o. In other words, we draw a random state (ω, s) and a goal requiring to move some object to some place p. We seek a two-step solution plan ⟨take, place⟩ to the problem: can the randomly chosen object o be picked from its initial pose, and can it be placed afterword on the required place p?

We call a motion-manipulation planner such as ManipPlanner for the take action and record its result as success or failure in this particular case. If the take is successful, we pursue the planning from the resulting state with a place action to ungrasp o in the required place p, and similarly record its success or failure result. This gives the following training set:

$$\mathcal{D} = \{((\omega, s, a), \text{result})| \text{ on simulated problems}(\omega, s, g), \text{result} \in \{0, 1\}\}$$

There is an infinite number of take instances for a given o, and similarly for place instances in a given p. To ease learning, we may partition the continuous set Q^o_{grasp} into classes of grasps. A natural partition for simple grippers and polyhedral objects is with respect to the object faces since a grasp is along the normal to a face. For box-like hexahedron objects, we'll have six possible classes of take and place instances. The precise grasping position $\mathcal{H}_{\boldsymbol{\theta},\mathbf{q}}$ along the normal to a face is randomly sampled by the ManipPlanner. We may also partition the continuous sets of Q^o_{sta} for the various places of the domain into coarse areas. This will give a discrete set of possible instances of place in randomly sampled poses in each area.

The training set generation would be more relevant for learning if it uses on the most frequent motion-manipulation problems for the application at hand.

Data encoding. As usual, the input state-action tuples (ω, s, a) need to be appropriately encoded as numeric vectors to use neural net approximators. A possible encoding for the state can rely on the geometric CAD model of the environment and objects. A more popular alternative uses directly the visual input of the scene. This alternative is appealing because simulators as well as real experiments can provide directly usable input encodings. However, a 3D image of a scene from a single perspective, e.g., from the top, may have partial or even total occlusions. There is a need to record the scene from several perspectives with e.g., five 3D images from the top, front, rear, left and right. In addition, we need to designate in these images (with image segmentation techniques) the object and goal place (or area) of interest in the training problem at hand.

The take and place instances can be encoded as One-Hot binary vectors taking into account the possible partitions of Q^o_{grasp} and Q^o_{sta}, but not the variables $\boldsymbol{\theta}'$ and $\wp$ sampled by ManipPlanner.

Neural net training. We have to architecte a neural net to be trained on $\mathcal{D}$ taking the encoded (ω, s, a) as input and giving as output a scalar $r_s(\omega, s, a) \in [0, 1]$ approximating the average success reward of a in (ω, s). Alternatively, we may output a vector whose components are the success rates $r_s(\omega, s, a_i) \in [0, 1]$ for the various classes of grasps and poses considered in the partition of Q^o_{grasp} and Q^o_{sta}.

The visual encoding of the input fits naturally with convolution neural nets. The learner can have a CNN channel for each of the 3D scene perspectives; their output are concatenated together with the action encoding and fed to a feedforward net for the final regression. Loss and triggering functions have to be chosen and the hyper parameters tuned for the domain.

Continual online learning. Learning on prior simulated data does not necessarily reflect the actor's working conditions and environment (see Section 16.1.4). It is desirable to focus the learning on the specific objects and environment of an application.

For that, we use prior simulation to initialize offline the learner, and rely on the estimated success reward function $r_s(\omega, s, a)$ to guide an F-TAMP planner for acting. We then use a procedure similar to CORL on recorded acting results to improve the learner towards a more precise success reward function.

Limitations. Recall that the probabilistic completeness of F-TAMP guided with the success reward fonction r_s, is conditioned on $r_s(\omega, s, a) > 0$ whenever a is feasible in (ω, s). Now, the targets for learning r_s come from the output of a probabilistically complete sampling-based metric planner. During the training set generation, we should make sure to get no a failure result when a solution exists. To avoid false failures in $\mathcal{D}$ we call the metric planner with long time-outs. This may not be sufficient; differences between the training environment and the runtime environment may still lead to rule out valid actions.

Indeed, the learner resulting from the procedure outlined here is specific to a class of similar TAMP domains. Changing even slightly the environment layout or the robot gripper would change significantly C_{free} or Q^o_{grasp}, thus the capabilities of the take and place actions.

Learning for a TAMP domains with a mobile manipulator, as Justin Figure 20.6(a), or DWR robots with container handling capabilities (as in Example 21.10) would require modeling other actions and acquiring simulation data about their success reward functions. As already discussed, RL allows generalizing across problems for the same domain, but what is learned does not easily transfer to other domains.

Finally, this approach for learning to guide a TAMP planner might not be very helpful in highly constrained manipulation problems (as in Figure 21.6). For that, extensions of F-TAMP with the methods of Section 21.3.3, possibly associated with learning sequences of motion-manipulation steps, might be more efficient.

22.3 Discussion and Bibliographic Notes

Let us first discuss skill learning then consider learning for TAMP.

Skill learning. It is mainly about motion and manipulation. The research area is rich with numerous significant contributions, for RL in general as well as in robotics. The general RL part has been discussed in Section 10.9. The robotics part is focused on policy-based techniques, surveyed in [622]. A broad survey covering RL approaches, imitation and transfer learning mostly for manipulation skills, is proposed in [646].

Most skill learning approaches have used model-based and policy-search methods. This is because the kinematic and dynamic models are often known in skill learning problems, with efficient methods for estimating the parameters. Moreover, policy search allows leveraging the domain knowledge with priors on the searched policies. Furthermore, a policy has often fewer parameters than a value functions.

Note that there is a close link between skill reinforcement learning and optimal control methods [130]. Both address the problem of finding a policy that optimizes an objective function (e.g., accumulated cost or reward), and rely on similar representations. Both have to handle partially observable states with e.g., filtering, expectation-maximization or probabilistic inference methods [699, 751], to estimate variable values as well as the uncertainly about these estimates.

A typical policy search technique for learning motor primitives is illustrated in [620]. The Deep Skill Learning algorithm is due to [718]. The *batch normalization* technique referred to earlier is described in [541]. These and similar approaches have been quite successful in games such as darts and archer aiming, tennis-table playing [891], or ball pushing, as demonstrated by the RobotCup competition winner [943].

Learning helicopter aerobatics flights, a very difficult task for human pilots, is an early success for RL in robotics [4, 6]. The approach assumes rigid dynamic models of the helicopter. It learns these models from the teacher's demonstrations, with improvement by reinforcement learning in autonomous flight. It then learns the reference trajectories of each aerobatic figure, starting also from the teacher's demonstrations for learning the reward function, as in inverse RL, to further optimize these trajectories. The control along the learned trajectories relies on receding horizon linear quadratic control. Experimental results demonstrated a wide range of impressive maneuvers.

Dexterous manipulation of a Rubik's Cube illustrates a recent success with an original approach called "*automatic domain randomization*" (ADR) [21]. It uses a vision-based state estimator neural net together with a recurrent neural net for policy reinforcement learning. Training is done on random simulations. The ADR method appears to play an important role for transferred learning.

Akin to transfer learning, let us mention the issue of multitask learning (discussed for general RL p. 262). Skill generalization has been addressed quite early, e.g., with meta-parameters [621], or skill trees [636], and more recently with shared policies [1084], or policy sketches [42]. Hierarchical policies and hierarchical RL methods are also actively explored for acquiring more general skills [827, 1201, 1004].

Learning for TAMP. Learning to improve the efficiency of TAMP planners starts naturally with the dominant run-time part in TAMP, that is the motion and manipulation part. Since this part rely on sampling in a continuous space, a natural idea, pursued by several authors, is to learn how to drive the sampling in beneficial areas of the C_{free}, Q^o_{grasp} and Q^o_{sta} spaces. Of notable interest is the approach of [533] which learns, from demonstrations of movements, a nonuniform sampling distribution using variational autoencoders.[1] The learned distribution is used to bias the sampling performed by the planner.

Most contributions on learning for TAMP focus on synthesizing feasibility heuristics of movements and grasps from prior planning. Approaches using neural nets with visual sensing as part of their input are for example [310, 1194, 1202, 19]. The latter is the basis of the description given in previous section; its gives performance

[1]This is a particular class of NN where an encoder net maps the input vector to a latent space corresponding to the parameters of a variational distribution, a decoder net maps the latent space to a vector output that follows the same distribution as the input.

improvement results with a forward search TAMP planner. It has been extended to multi-robot TAMP problems and objects with various shapes [20].

The approach of [1166] trains an SVM supervised classifier discriminating feasible from unfeasible moves. Training is performed in the metric feature space on simple environments with just two objects, generalized later. The approach uses an incremental constraint-based TAMP planner [272], which can go beyond the recommandation of the trained classier to guarantee probabilistic completeness.

The contribution of [230] considers TAMP as metric refinements of symbolic skeleton plans. It tackles the metric refinement stage with reinforcement learning on an MDP whose nodes are skeleton plans and whose edges are selections (by sampling) of metric refinements. A reward is the fraction of the steps in the skeleton plan that have satisfied metric precondition. The method relies on inverse reinforcement learning (see Section 10.7.3) from expert demonstrations to guide the learning.

Akin to inverse RL, the approach of [1062] relies on learning from demonstration to synthesize a manipulation graph annotated with multimodal sensing data for a robust manipulation. It is able to tackle assembly tasks with grasping, unscrewing, and insertion.

Note that the Rubik's Cube approach of [21] referred to earlier, can be considered as going beyond learning a sensory-motor skill, since it requires manipulation as well as solving the cube. The ADR method should possibly be generalizable to TAMP. But since learning is end-to-end (without a planner) it is unclear how flexible the result can be for a diversity of problems in the same TAMP domain.

22.4 Exercises

22.1. Consider a simple pick-and-place action. Analyse under what conditions a skill learned by Deep Skill Learning would be sufficient for performing this action and when a manipulation algorithm such as ManipPlanner would be needed. How may the two approaches be combined.

22.2. The learning approach for TAMP problems presented in Section 22.2 provides control knowledge for a forward search TAMP planner such as F-TAMP. How this knowledge could be of use to the HTN-like TAMP planner of Section 21.3.4

22.3. The "Informed metric backtracking" algorithm for TAMP of Section 21.3.3 relies on the notion of culprit actions violating different types of constraints. Devise a learning method, akin to the method described in , to synthesize control knowledge for an "Informed metric backtracking" TAMP planner.

Part VIII

Other Topics and Perspectives

The end of a melody is not its goal: but nonetheless, had the melody not reached its end it would not have reached its goal either.

Friedrich Nietzsche, *The Wanderer and His Shadow*, 1880

Previous parts of the book covered acting, planning and learning for different types of models. There are several deliberation functions needed by an autonomous actor that do not naturally fit into these seven parts, but should not be totally ignored.

Moreover, AI research is going through fast-moving and highly connected transformations. In the past, techniques for natural language translation were not very relevant for acting and planning systems. With the recent advent of Large Language Models and their various multimodal extensions into Foundation Models, this is no longer the case.

This last part of the book briefly surveys these topics. Chapter 23 introduces Large Language Models and their potential benefit in acting, planning and learning. Chapter 24 discusses the perceiving, monitoring and goal reasoning functions for deliberation.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

23 Large Language Models for Acting and Planning

Methods for addressing Natural Language Processing tasks (NLP), such as text comprehension, translation, summary, or dialogue, have always been of much concern to AI. However, they were usually decoupled from those of acting and planning.[1] NLP methods, per se, are not within the scope of this book. However, the recent developments of *Large Language Models* (LLMs) and their extension to multimodal *Vision-Language Models*, *Action-Language Models* and *Foundation Models* have introduced a radical change.[2] This change is reflected in most of the AI journals and publication venues. For example, the "AI Index" [758], a comprehensive report on the state of the field, devotes its 2024 edition for the frontier of AI research almost solely to these systems.

An LLM is basically a very large neural net trained as a statistical predictor of the likely continuation of a sequence of words. LLMs have excellent competencies over a broad set of NLP tasks. Additionally, LLMs demonstrate the emergence of deliberation capabilities for reasoning, common sense, problem solving, code writing, or planning. These abilities have not been designed for in LLMs. They are unexpected and remain to a large extend poorly explained. Although error-prone and imperfect, they open up opportunities for automated deliberation that draw tremendous research

Section 23.1 introduces the reader to LLMs. Section 23.2 discusses LLMs features with respect to planning, acting and learning. This brief chapter aims to clarify the fundamental issues; the corresponding techniques are changing very quickly.

23.1 Principles of LLMs

Consider the following prediction problem:

> given a *context* expressed as a sequence of observed variables $\langle x_1, \ldots, x_{n-1}\rangle$, predict a likely next term x_n.

When the x_i are the states of a well modeled system, one may address this problem with the model of the system under consideration. This has been illustrated earlier, e.g., at the movement and control level with Kalmann filtering methods, or at the abstract causal state transition level with Markov models. Weather forecast and numerous state prediction processes illustrate this generic problem.

[1]Except for issues such as dialogue planning.

[2]The word 'model' is a misnomer for these systems. In science, a model of some reality is expected to be intelligible, explanatory, justificatory, and predictive. Here, only the latter is addressed in a statistical sense. We keep however the acronym LLM and use it also for the multimodal cases.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

When no principled model is available but the domain is not too erratic, one resorts to statistical prediction. This fallback option produces a "shallow model", which makes predictions based solely on statistical reasoning, not on explicit causal relations. As a consequence, such a model makes predictions about a system's behavior without attempting to explain how the system operates. Statistical prediction assumes that the domain is *regular*: this is the *induction assumption* stating that the statistical relations observed in the training data hold also for unobserved cases. It also requires that sufficient training data can be acquired.

A simple and familiar example of statistical prediction in NLP is the typing help one finds in many devices. In NLP, training data is plentiful.[3] Moreover, natural language is regular, as measured by its low entropy. This is known since the introduction of information entropy by Shannon: "*Anyone speaking a language possesses, implicitly, an enormous knowledge of the statistics of the language* (... that enables) *to complete an unfinished phrase in conversation*" [1001]. Shannon proposed the notion of "N-gram", which is the above prediction problem applied to sequences of letters or words with the estimation of the conditional probabilities $\Pr[x_n \mid x_1, \ldots, x_{n-1}]$, x_i being the i'th letter or word in the sequence.

N-grams do not inform on the semantics of words. NLP tasks require to assess which words are semantically close. Two words a and b are likely to be semantically close if they are used in similar contexts, that is, for various sequences $\langle x_1, \ldots, x_{n-1}\rangle$ we have $\Pr[a \mid x_1, \ldots, x_{n-1}] \simeq \Pr[b \mid x_1, \ldots, x_{n-1}]$. This approach faces a complexity issue: the conditional probability table $\Pr[x_n \mid x_1, \ldots, x_{n-1}]$ has size $O(\delta^n)$ where δ is the size of the dictionary. Typically δ is about 10^5 words, and a context $\langle x_1, \ldots, x_{n-1}\rangle$ may contain several hundreds words.

This complexity issue has been addressed with the "neural probabilistic language model" [111], the basis of *word embeddings*. Embeddings are mappings from the set of words or sentences to a metric space, $\mathbb{R}^d$, in which semantically similar words have proximate embeddings, i.e., they are mapped to points close in $\mathbb{R}^d$ for a metric function or distance. Word embeddings can be computed with neural nets and used in NLP tasks by other nets. Simple vector calculations are used in NLP operations.

Example 23.1. Let $[\overrightarrow{word}] \in \mathbb{R}^m$ be an embedding of *word*. The word embedding may enable arithmetic operations such as:

$$\begin{array}{ll}
\text{relationships:} & [\overrightarrow{France}] + [\overrightarrow{capital}] \simeq [\overrightarrow{Paris}],\\
\text{analogies:} & [\overrightarrow{Berlin}] - [\overrightarrow{Germany}] + [\overrightarrow{Japan}] \simeq [\overrightarrow{Tokyo}],\\
 & [\overrightarrow{copper}] - [\overrightarrow{Cu}] + [\overrightarrow{gold}] \simeq [\overrightarrow{Au}],\\
\text{proximity:} & [\overrightarrow{speak}] \cdot [\overrightarrow{converse}] < [\overrightarrow{converse}] \cdot [\overrightarrow{gossip}].
\end{array}$$

Here "$\cdot$" is the dot product, which is null when the two vectors are orthogonal and maximal when they are aligned. □

For a while, Recurrent Neural Nets (RNNs) have been used for NLP. In RNNs outputs from hidden layers are fed back as inputs. RNNs provide a limited handling

[3] Web-accessible documents are estimated to more than 10^{12} words.

of sequences of words. They cannot deal with long sentences, because significant links in sentences may not be between consecutive words. Memory-augmented variants of RNNs, such as Long Short-Term Memory (LSTM) nets, have been used in NLP and in other tasks to handle dependences between distant terms in sequences. However, NLP faces numerous challenging ambiguities such as co-reference resolution and matching of pronouns. These require a good grasp of the semantics and pragmatics of the language, possibly over a long context.

Example 23.2. Consider these two sentences: *(i) The dog did not cross the stream, it was too deep*; and *(ii) The dog did not cross the stream, it was tired.* The pronoun "it" refers to the stream in *(i)* and the dog in *(ii)*.

The following illustrates an additional language pragmatic ambiguity: *(i) Sara could not board the boat, she was late*, and *(ii) Sara could not board the boat, she was over crowded.* Sailors (and old english) refer to boats with feminine pronouns. □

NLP ambiguities have been successfully addressed with "attention" mechanisms that provide ways to relate words at different positions in a sequence. These mechanisms often use an encoder-decoder architecture. An encoder maps an input sequence of terms $\langle x_1, ..., x_n \rangle$ to another sequence $\langle z_1, ..., z_n \rangle$, from which the decoder generates an output sequence $\langle y_1, ..., y_m \rangle$. Each step takes as additional input the term generated in the previous step (this is called an "*auto-regressive*" model). The input/output terms are embedding of words, or more precisely of fractions of words called *tokens*.

Significant progress has been obtained with the multi-head attention transformer architecture [1119]. Transformers consist of alternating attention nets and feedforward neural nets. The former uses efficient parallelized processing to assess the relative weights between all pairs of tokens in a context. Multi-heads allow for disambiguation of the meaning and relations between words: the learned relative weights allow relating "it" to "stream" in-sentence *(i)*, and to "dog" in sentence *(ii)*.

LLMs are pre-trained by self-supervised learning on numerous documents from the internet. Self-supervised means that the system learns to predict the next word in the documents it reads. In addition, many LLMs offer a chat or a dialogue interface. Further training steps are used to avoid possibly undesirable dialogues and align LLMs with human preferences. These steps are, for example, Reinforcement Learning with Human Feedback (RLHF) to automatically provide a kind of reward shaping [861] (see also Section 10.7.1), or with rule-based reward models [427]. Furthermore, an LLM can be tuned to specific applications and/or adapted within a dialogue to a particular task through a few prompts (called *in-context learning*).

LLMs are not endowed with formal knowledge models. They do not know about grammar or logic. They have no implemented algorithmic or reasoning capability. They cannot do search. Except for the learned parameters, they have no memory to store data structures. Their sole computational mechanisms are back propagation for training and forward propagation for predicting, according to the net topology.

Transformers, differently from RNNs and LSTMs, have no recurrent units. On the one hand, this is an advantage: they require (in principle) less training time than RNNs and LSTMs. On the other hand, they are limited in expressiveness with respect to RNNs and LSTMs, as demonstrated in [460]: they cannot implement recursion.

LLMs face several theoretical limitations. For example, they cannot correctly handle non-regular context free languages, recursive languages or regular periodic languages, which are much simpler than the natural language [460]. It has been shown that training with a finite number of textual contexts, which provide clues about the underlying semantics of an assertion, cannot provide an "understanding" of a language in the sense of a formal denotation semantics [785]. However, LLMs are able to capture some important aspects of meaning, such as "conceptual roles" according to conceptual role theory [785].

These theoretical limitations are in a way similar to worst case complexity results, which do not preclude practical performances.[4] LLMs in practice have significantly improved the state of the art and have demonstrated proficiency in most NLP tasks. Although an LLM can't handle non-regular languages in general, a large enough LLM can handle a large but finite subset of a language, since a finite subset can always be expressed as a regular language. The larger the subset one wants an LLM to handle, the larger the LLM must be, in order to learn all of the non-regular special cases.

An LLM's chat interface leads one naturally to query a model like an oracle capable of answering anything, including for issues outside of its scope. LLMs are criticized as being non-factual. They are said to "hallucinate" or "confabulate". These terms are rather misleading with respect to the fundamentals of statistical induction: for a pair (x_i, y_i) in a training data base, a neural net function may not give $f(x_i) = y_i$ since f is a statistical approximation function, not a database query.

LLMs have demonstrated limited but surprising abilities over a broad set of cognitive tasks for which they have not been designed. These tasks range from arithmetic, programming, or logic reasoning, to common sense and planning. Arithmetic for example is totally unexpected from a statistical approximation model: one does not learn arithmetic from the statistic of computations, but from the synthesis of specific algorithms. Have LLMs been able to synthesize such algorithms in some way?

Empirical observations show a clear scale effect: below some size of the network (about 10^{10} parameters) these abilities are nonexistent, while above this threshold they grow significantly with the network size [1160]. Several conjectures are being investigated about an LLM capability to synthesize, in some form, adapted procedures to deal with a task. For example, an LLM trained on a database of Othelo board games (a straight LLM sequence generator, without search nor RL as in, e.g., Alpha-Go) is able to generate legal move and play at a modest level. Probes of the learned parameters (a technique inspired from neurology) seem to reveal the board topology and the game transition function in the trained network [707]. More investigations about LLMs fundamental abilities are needed to characterize these computational models.

On the practical side, significant advances are being made and already address some of the initial limitations, for example:

- Limited context: a system like Anthropic Claude3 claims to handle a context of up to 10^6 tokens, i.e., a full book [46].

[4] Planners perform reasonably well on PS-Space hard problems.

- Interaction with specialized solvers and tools: for example, ChatGPT has been interfaced with mathematical solver Wolfram|Alpha. [5]
- Dynamic adaptation and incremental learning: this is being addressed with e.g., Instruction Tuning, In-Context Learning, Chain-of-Thought, Tree-of-Thought and other methods [1007, 561].
- Justification and reference to source material: the *Retrieval Augmented Generation* (RAG-LLM) approach opens promising capabilities [223, 379, 555].

LLMs face two bottlenecks for their further development:

- the availability of larger training bases, and
- the energy cost and entailed climate footprint for their training and use.

On the former, most available open source documents have been used for pre-training current LLMs.[6] Larger models would require larger training bases. The energy and footprint bottleneck relates to the computational complexity of an LLM, which depends on many features of their architecture, such as the number of attention layers (about 100) and total number of parameters (about 10^{11} to 10^{12}). Basically the complexity of a prediction step in an LLM is in $O(n^2 d)$, where n is the length of the context (about 10^4 to 10^6 tokens) and d the dimension of the embedding space (about 10^5 in recent implementations) [1119]. Pre-training is in $O(mn^2 d)$ where m is the size of the training base. Empirically, m is of the same order as the number of network parameters (i.e., about 10^{11} to 10^{12} tokens).

Despite active research to reduce this complexity (e.g., to $O(nd)$ per prediction step without too much loss in performance [857]), the LLM technique remains very expensive. It has been estimated that GPT3 training required 1.3 GWh [874].[7] Several empirical models have confirmed these estimates (e.g., thousands Joules per step for the LlaMa system which has "only" 65G parameters [975]). Various optimizations such as energy capping and scheduling bring a few percent savings [775], but do not change the fundamentals. With current approaches, increased performances are expected to cost significantly more.[8] More frugal approaches are definitely needed; they start to be the topic of active research, e.g., [315, 557].

Beyond this brief overview of a fast moving area, the reader is invited see the works surveyed in [1229, 834], and [527] for a focus on reasoning in LLMs. The briefly mentioned Multimodal Foundation Models, which are trained with and handle text, images, speech and heterogeneous data, are surveyed in [151, 704].

23.2 LLMs in Acting, Planning and Learning

In this section we discuss a few approaches leveraging LLMs capabilities for planning, acting, and reinforcement learning.

[5] See https://writings.stephenwolfram.com/

[6] GPT3 has been trained on about 10^{11} words, out of the 10^{12} to 10^{14} estimated on the web.

[7] This is about the average monthly needs of a town of over 7000 persons in France.

[8] It has been estimated that a 10-fold improvement in model performance of deep learning comes at a cost of a 10,000-fold increase in computation and energy [1094].

23.2.1 LLMs and planning

Several studies (e.g., [1111]) have used LLMs on classical planning benchmarks, such as blocks world, and reported that they do not compete with a good planner. However, these are not very informative and fair comparisons. A planner is highly specialized; it is given a well formalized problem specification and the corresponding knowledge. An LLM is given an informal, partial problem specification that requires significant common-sense knowledge. It is not obvious how an LLM can give a reasonable solution to a planning problem, as in the following example.

Example 23.3. In [187], an early version of GPT4 was prompted with the following problem: "*Here we have a book, 9 eggs, a laptop, a bottle and a nail. Please tell me how to stack them onto each other in a stable manner.*" The model responds with a reasonable plan, including numerous details about how to perform the plan, with common-sense recommendations about stability and fragility issues. □

No classical planner can handle an informal, sparse specification, as in Example 23.3. The formal specification of this example would require significant efforts. Planners are narrow; they cannot do text translation, summary, Q/A, or prove (in verse) that the set of prime numbers is infinite [187]. A comparison of LLMs *vs* planners and an assessment of how LLMs can be used in planning should take into account their features, strengths and weaknesses.

The main features of LLMs *vs* planners can be sketched as follow (see Figure 23.1):

- *Broad versus narrow knowledge.* Planners work on specific domains, while LLMs are pre-trained with a huge set of bulk data (text, or even other heterogeneous data if they are multi-modal foundation models) covering different areas, e.g., medicine, religion, history, humanities, science, law. They show impressive capabilities in all these areas. Plan generation and learning techniques can be general but, even in the case of generalized planning [631, 1052], they cannot deal with so broad spectrum of areas. No surprise if LLMs have been applied to generalized planning [1021].
- *Self-supervised training versus Human specification of knowledge.* LLMs learn mostly from unsorted available documents automatically crawled over the web. Planners require careful and formal specification of a domain knowledge.
- *Shallow versus deep knowledge.* Planning performs extended search with action specifications that describe abstract causal state transitions. LLMs are simple statistical predictors.
- *Opaque versus explainable process.* Statistical induction is based on a regularity assumption. The corresponding approximation has no causal support. Hence LLMs cannot explain why some data/event/situation entail other data/event/situation. Planners build and can easily exhibit causal chains.
- *Error prone versus correct/provable.* LLMs are large approximation functions. They can be incorrect in all cases where the statistical induction leads to mistaken generalization. Planning is provably correct with respect to the given model of actions and states. This does not implies correctness with respect the

real world, since the model can be an incorrect or incomplete formal representation of the world. However, models can be verified up to some assumptions, and correctness with respect a model is a requirement for safety critical applications.

- *Scalable versus not scalable.* LLMs can deal with intuitive domain specifications that would require huge formal representations in planning. An LLM prediction step is of polynomial complexity in the size of its input, while planning is PSPACE-hard. Scalability has also to take into account that LLM training is mostly self-supervised, while planning domain specification is manual.
- *Generalizable versus not generalizable.* Pre-trained LLMs can be fine tuned to different domains. Most of the plan generation techniques are domain independent; a model can hardly be transferred to different domains.
- *Informal versus formal input.* LLMs have beed designed to interact in natural language. The input to a planner is formal.

Note that these comparisons are not specific to LLMs *vs* Planners. Most of them also hold for LLMs *vs* solvers and formal reasoning systems.

Figure 23.1. Main features of LLMs *vs* Planners.

LLMs	*Planners*
Broad, huge repertoire of knowledge	Narrow knowledge
Self-supervised training	Human specification of knowledge
Shallow reasoning	Deep reasoning
Opaque process	Explainable
Error prone	Correct, provable
Generalizable to many domains	Not easily generalizable
Scalable	Barely scalable
Informal, natural language I/O	Formal I/O

In summary, planners work on specific domains, while LLMs are pre-trained with all sort of documents (texts and other heterogeneous data if they are multi-modal foundation models) covering many areas. Planning, even generalized planning, are not as general as LLMs. Clearly, a desirable perspective would be to mix the good properties of both approaches.

To leverage the capabilities and desirable features of LLMs in planning, several strategies are being explored, among which the following:

- *Adapt* a pre-trained LLM to planning with additional specific training, in-context learning and prompting [202, 864, 865, 866]. For example, the "chain-of-thought" prompting uses a few instructions or steps to decompose complex tasks in order to progressively guide the LLM [1151]. The "tree-of-thought" generalizes this to a branching interaction [1203].
- *Train* specifically LLMs as planners [202] or generalized planners [1021].
- *Delegate* a planning problem: formalize it, identify its type and call an appropriate planner to solve it [726, 267].

- *Interface* a planner with an LLM in human-machine interactions, to model human mental states and grasp human reasoning [1134].
- *Acquire* with LLMs planning domain models and control knowledge:
  - *Translate* with LLMs informal NL descriptions to formal planning domain specifications [226, 451, 1189]. Different LLMs can generate correct planning domain from NL descriptions [860].
  - *Sketch* with LLMs skeleton plans to be used as heuristics by a specialized planner [179].

These strategies are not mutually exclusive. They may be mixed in various ways, in particular with the approaches discussed next in LLMs for acting and learning. There are other strategies that seek to endow LLMs with short-term and long-term memories to improve deliberation capabilities [1227]. LLMs can also be applied to specific planning problems such as navigation [932], multi-agent planning, interactive planning, heuristics optimization [866], and learning general policies [962].

Note that the "*Delegate*" strategy corresponds to a general objective to interface an LLM with a library of specific tools, e.g., for computation, search or planning, and train it to use this library appropriately [868, 988, 709]. Here, an LLM extends its broad informal knowledge with models of specialized computational means, and its shallow common-sense reasoning with the functionalities of its tools. Since LLMs are used successfully for code generation, e.g., [187], this may possibly go up to acquiring some generic computational model and a capability of generating programs for tasks not covered by an available library.

These research perspectives for planning with LLMs are promising but they remain today preliminary. Most contributions are empirically validated on limited domains. Critical issues such as correctness are usually not addressed, or quite partially with a few tests. Clearly, more investigation is needed for LLMs in planning.

23.2.2 LLMs and acting

Recall that acting involves getting feedback during an activity and adapting actions to this feedback. The purpose here is to leverage LLMs in refining actions into executable commands with feedback from an execution platform or a simulator.

Some approaches use LLMs only for the synthesis of a high level plan (using the strategies of the previous section) and rely on a specialized low-level planner to effectively refine and execute actions [1222]. Other approaches plan with LLMs and keep refining generated plans until the primitive executable level [530].

Many contributions on LLMs for acting are concerned with robotics, e.g.,

- for the synthesis of programs controlling situated robot tasks [1029, 712];
- for grasping and manipulation problems [1076];
- for task and motion problems [720];
- for the synthesis of a visual servoing controller from the informal description of a computer vision library [1127];
- for training an LLM to generate behavior trees for a hierarchy of tasks from a user description [200].

Other contributions to LLMs in autonomous agents are surveyed in [1150].

The generic “embodied LLMs” (or foundation models) approaches seek to incorporate real-world continuous sensor modalities into language models and to link words to percepts. An illustration is the PaLM-e system which injects images and multimodal information into the embedding space of a pre-trained LLM [311]. It has been tested of a few robotics manipulation tasks demonstrating an interesting task-transfer capability.

Note that LLMs and their multi-modal versions are potentially very adequate for acting with human feedback and for human-robot interactions [1221]. They offer essential functions well beyond NL and oral interactions, such as intent estimation and theory of mind reasoning [1147].

23.2.3 LLMs and reinforcement learning

Many investigations use RL in the LLM training process to improve their performances in NLP tasks, and make the generated texts safer and aligned with human preferences. Among these studies, for example, RL is used to extends the auto-supervised pre-training phase with human feedback [859], or with an automated and more scalable procedure [690]; alternatively, RL can be used as a dataset generator for fine-tuning an LLM training [450].

More relevant to the focus of this book are approaches in the opposite direction that use LLMs for RL. These approaches, sometime referred to as *language-conditioned RL*, seek to leverage LLM capabilities to help an actor efficiently learn optimized behaviors across numerous tasks. Most of the issues about *aided* RL (covered in Section 10.7) are possibly relevant for an LLM support. Recall that the reward function is critical for RL: it expresses the task and guides learning. But of simple cases, an informative reward function is not obviously entailed from the task. Human guidance is needed, for which the common sense and broad knowledge of LLMs can be of help.

Reward shaping from the specification in natural language of a user’s preferences is a first natural approach, investigated in, e.g., [201, 442]. The synthesis of the reward feedback with an LLM has also been explored [663, 744, 1188]. The LLM is informed through user’s prompts about the environment, the task at hand, and possibly about her assessments of a few observed behaviors [1046]. One step further seeks to ease RL task transfer with the LLM synthesis of abstract actions, intermediate auxiliary tasks, and exploration guidance [313, 924]. Most of these approaches have been tested in robotics tasks, often in simulation, or in game playing, e.g., Mindcraft [1149].

Another class of approaches uses LLMs at the policy level, for the synthesis of well informed priors [523], the generation of sequences of actions [224, 273], or even as the policy to be updated [205]. Other authors use LLMs to plan a high level behavior combining skills learned with RL [66, 17], or use RL to improve the planning capabilities of an LLM [1008]. Other approaches include:

- leveraging LLMs for hierarchical RL [554], and learning hierarchical policies from unannotated demonstrations using a known library of skills [1005];

- imitation learning to train an LLM from demonstration generated by a task and motion planner in vision-based manipulation tasks [270];
- reward shaping for RL in sparse reward domains in human-AI collaborative applications [795].

Additional approaches are discussed in the survey and taxonomy of LLM versus RL methods of [918].

Finally, recall the proposed use of RL for learning refinement methods in a partial programming paradigm (Section 16.2). Since the body of a refinement method can be any program, and LLMs can be used as programming assistants when adequately pre-trained [1197], the synthesis of partial program by LLMs, further refined with RL and additional guidance, can open interesting perspectives.

24 Perceiving, Monitoring and Goal Reasoning

Acting, planning and learning are critical cognitive functions for an autonomous actor. Other functions, such as perceiving, monitoring and goal reasoning, are also needed and can be essential in many applications.

This chapter briefly surveys a few such functions and their links to acting, planning and learning.[1] Section 24.1 discusses perceiving and information gathering: how to model and control perception actions in order to recognize the state of the world and detect objects, events, and activities relevant to the actor while performing its tasks. Section 24.2 is about monitoring, that is, detecting and interpreting discrepancies between predictions and observations, anticipating what needs be monitored, controlling monitoring actions. Goal reasoning in Section 24.3 is about assessing the relevance of current goals, from observed evolutions, failures, and opportunities for achieving a higher level assigned mission.

24.1 Perceiving and Information Gathering

Acting requires knowing one's environment, how is it structured, what it contains, and where relevant objects might be. An autonomous actor cannot be too dependent on predefined knowledge given *a priori*, which is generally costly and brittle. Moreover, in an open environment an actor can only have a partial knowledge about its environment. It has to perceive what is relevant for its activity and to reason about its perception while performing its tasks. This leads to numerous problems about:

- Reasoning on sensors: how and where to best use a sensor, or to query relevant information, how to handle sensor changes and perform active perception.
- Reasoning on signals: interpretation, data association, symbol grounding and anchoring.

These problems have to be addressed in the context of wider issues such as:

- Reliability: assess how reliable are perception and information gathering actions. What verification and confirmation steps are needed to confirm that a sensed value of a state variable is correct? How to assess the distribution of values if uncertainty is explicitly modeled?
- Observability: how to acquire information about non-observable state variables from the observable ones? How to balance costly observations with approximate estimates?

[1] Additional contributions, more specific to robotics, are surveyed in [539].

Free pre-publication, for personal use only. To be published by Cambridge University Press.

- Persistence: for how long a state variable may keep its previous value when no new observations contradict it?

In the following, we focus on perception problems in deliberation. The signal processing issues, although important for perception, are not within our scope. We briefly survey a few approaches to *(i)* planning and acting with information gathering, *(ii)* planning sensing actions, *(iii)* perceiving for semantic mapping, *(iv)* anchoring and signal-to-symbol matching problems, and *(v)* recognizing plans and situations.

24.1.1 Planning and Acting with Information Gathering

The *closed-world assumption* (assuming that facts not explicitly stated are false)[2] is too restrictive. A deliberative actor lives in an *open world*. It has to handle partially specified instances of a domain and extend its knowledge when needed. In particular it needs the following capabilities:

- Plan with respect to objects and properties that are unknown when planning starts but that can be discovered at acting time through planned information-gathering actions. New facts resulting from these actions are used to further refine the rest of the plan.
- Query databases for facts the actor needs specifically to address a given planning problem and query knowledge bases for additional models of its environment that are relevant to the task at hand.

Planning with information gathering is studied by several authors using conditional planning approaches, as in the PKS system [894]. The continual planning approach of MAPL postpones part of the planning process [177]. It introduces information-gathering actions which will later allow development of the missing parts of the plan. The planner uses assertions that abstract actions to be refined after information-gathering. The approach is adapted to dynamic environments where planning for subgoals, that depend on yet unknown states, can be delayed until the required information is available through properly planned information gathering actions.

Acquiring additional data and models at planning time is useful in *semantic Web* [501]. For example, the ObjectEval system acquires from the Web statistics about possible locations of objects of different classes [973]. It uses them in a utility function for finding and delivering objects in an office environment. Other approaches use Description Logic (DL), a fragment of first-order logic, to handle statements about objects, properties, relations, and their instances with inference algorithms for querying large stores of data and models over the Web [65]. Most approaches rely on OWL, the Web Ontology Language, to partially handle an open-world representation.

24.1.2 Planning to Perceive

A perception action may not be always executable. Sensor models are needed to decide where to put a sensor, how to use it, how and when to best acquire the needed information. Planning to perceive is about integrating the selection of viewpoints

[2]A less restrictive assumption takes facts not entailed from explicit statements to be false.

and sensor modalities to a plan. It relates to the sensor placement problem, which is usually addressed as a search for selecting the next best viewpoint in tasks such as modeling an environment or recognizing an object, e.g., [683].

The integrated sensor placement and task planning problem is sometimes addressed with POMDPs, for example in [898, 916]. The HiPPo system [1048] uses hierarchical POMDP for sensor placement in the recognition of objects on a table, as typically required in a manipulation task.

An alternative and more scalable approach for synthesizing an observation plan within a navigation task seeks to detect and map objects of interest while reaching a destination [1122]. The approach uses a Bayesian method to correlate measurements from subsequent observations and improve object detection; detours are weighed against motion cost to produce robust observation plans using a receding horizon sampling scheme. The approach was tested in an indoor environment for recognizing doors and windows.

24.1.3 Perceiving for Semantic Mapping

An actor needs to know its environment, i.e., to have a map with the informations needed for its tasks. Section 20.3 already addressed environment mapping issues for navigation tasks, but mostly at the metric level. Here we briefly review exploration and mapping at the semantic level. We would like an actor to be able to find out where relevant objects of interest are, how do they look, how to retrieve them, what are the categories of various areas in the environment and their properties, e.g., a kitchen (where cooking appliances are), a living room, a study room, etc.

Computer vision and image recognition have made significant progress. The state of the art with supervised learning methods allows training a vision system for feature learning, segmentation and robust recognition in complex images of human labelled objects and places, e.g., [922, 524], or [950] in 3D scenes. But this supervised training remains costly and quite specific. Few-shot methods seek to recognize sparsely seen objects with the help predefined knowledge, e.g., [525]. The reliance on domain specific knowledge can be an advantage for untypical and relatively stable environments, but in general this dependance is a bottleneck.

Vision-Language Models (VLMs) open a promising class of approaches for semantic mapping. VLMs are an instance of the *Foundation Models* mentioned earlier. They extend LLMs to multi-modal data [414, 1223], but they are more concerned with image understanding than image generation. An illustration of VLMs is the *Contrastive Language-Image Pre-Training*(CLIP) method [927]. Since they are trained on large repertoires of image-text pairs, VLMs allow addressing semantic object labelling and environment mapping with an open vocabulary [413, 703, 1195], including for 3D scenes [884].

Promising hybrid approaches combine VLM techniques with spatial reasoning, using a specific solver, to ground the symbols with perception data [1063]. They address part of the anchoring problem, discussed next.

24.1.4 Anchoring

Acting and perceiving take place at the sensory-motor level but require reasoning and planning at an abstract symbolic level. Anchoring is about creating and maintaining over time a mapping between the two levels, i.e., between symbols and sensor data that refer to the same *physical object*. It can be seen as a particular case of the *symbol grounding* problem, which deals with broad categories, for example, any "door" as opposed to a specific one. It relies on the recognition techniques just discussed.

Anchoring is achieved by establishing and keeping a link, called an *anchor*, between the perceptual system and the symbol system, together with a signature that estimates some of the attributes of the object it refers to [256]. The model of an anchor relates relations and attributes to perceptual features and their values.

Establishing an anchor is a pattern recognition problem. The challenge is to handle the sensing uncertainty and the models ambiguity. Both can be dealt with, for example, by maintaining multiple hypotheses. This is illustrated in [586], which handles ambiguous anchors with a conditional planner to explore a space of belief states representing the incomplete and uncertain knowledge due to partial matching between symbolic properties and observed perceptual features. The approach distinguishes between definite symbolic descriptions, which are matched with a single object, and indefinite descriptions. Actions have causal effects that change object properties. Observations can change the partition of a belief state into several new hypotheses.

A dynamic probabilistic anchoring method has been proposed in [1205]. It relies on a multiple hypothesis tracker with a solver reasoning on knowledge given *a priori* about the domain and the context. The solver seeks to reduce the anchoring uncertainty. The approach has been integrated in a robot platform using a color camera. Experiments demonstrate a capability of resolving anchoring ambiguities in difficult cases, e.g., to distinguish identical instances of two objects from the scene context and relations such as contains or is-inside with respect to other objects.

Not every perceived feature needs to be anchored. The actor has to choose which anchors to establish, and when and how. Anchors are needed for all objects relevant to the actor's activity. Often, these objects cannot be defined extensionally (by specifying a list of objects). They must be defined by their properties in a context-dependent way. Object recognition is required not only to label specifically queried objects, but also to discover and anchor objects relevant to the task. Here VLMs can bring significant advantages for open environments and vocabularies.

Tracking anchors is another issue, *i.e.*, taking into account object properties that persist across time or evolve in a predictable way. Predictions are needed to check that new observations are consistent with the anchor and that the updated anchor still satisfies the object's properties. Finally reacquiring an anchor when an object is re-observed after some time is a mixture of finding and tracking. If the object moves, it can be quite complex to account consistently for its behavior.

The DyKnow system [488] illustrates several of the preceding capabilities. It offers a comprehensive perception reasoning architecture integrating different sources of information, with hybrid symbolic and numeric data at different levels of abstraction, with bottom-up and top-down processing, managing uncertainty, and reasoning on explicit models of its content. It has been integrated with a planning, acting, and monitoring system [301] and demonstrated for the control of UAV rescue and traffic surveillance missions. In the latter, a typical anchoring task consists of recognizing a particular vehicle, tracking its motion despite occlusions, and re-establishing the anchor when the vehicle reappears, e.g., after traversing in a tunnel.

Finally, we note that anchoring problems can be addressed with the embodied LLMs approaches mentioned in previous chapter. The PaLM-e system [311], which integrate real-world continuous sensor modalities into language models, is able to establish anchors between words to percepts.

24.1.5 Event and Situation Recognition

The dynamics of the environment is an essential source of information, as we just saw in the anchor tracking and re-acquiring problems. An actor needs to comprehend what an observed sequence of changes means, what can be predicted next from past evolutions. This is essential for interacting with other actors, to understand their intensions and behavior, for example, in robot tutoring [50], or in surveillance applications [518, 377].

Several contributions to action and plan recognition are surveyed in [647]. They deal with *(i)* human action recognition, *(ii)* general activity recognition, and *(iii)* plan recognition. The former two types of processing provide input to the latter. Most surveyed approaches rely on signal processing and plan recognition techniques. The former use filtering approaches, Markov Chains, Hidden Markov Models and neural nets [553, 635, 1064]. They have been applied to movement tracking and gesture recognition by [1182, 800]. Plan recognition rely on deterministic planning approaches of [590, 933], or probabilistic approaches [395], as well as on parsing techniques of [921].

Many plan recognition approaches assume as input a sequence of symbolic action labels. This assumption is hard to meet in practice. Usually actions are sensed through the observation of movements and their effects on the environment. The recognition of actions from their effects depends strongly on the plan level. Decomposing the problem into recognizing actions then recognizing plans from these actions is fragile. More robust approaches have to start from the observation of changes.

Chronicle recognition techniques are relevant to this problem. Chapter 17 defines a chronicle is a model for a collection of possible scenarios. It describes classes of events, persistence assertions, non-occurrence assertions, and temporal constraints. A ground instance of a chronicle can be formalized as a nondeterministic timed automata. Beyond planning operators, chronicles can be used to describe classes of dynamic situations and plans, and to recognize their occurrences from observations [404, 308]. The approach monitors a stream of observed events and recognizes, on the fly, instances of planned chronicles that match this stream. The recognition is efficiently performed by maintaining incrementally trees of hypotheses for partially recognized chronicle instances. The trees are updated or pruned as new events are observed or time advances. It has been demonstrated in robotics surveillance tasks. Other developments introduced hierarchization and a focus on rare events [307].

The chronicle approach offers a link between planning and observing, e.g., what needs to be recognized can be planned for in advance. The SAM system [878] illustrates such in a system providing assistance to an elderly person. It uses a chronicle representation (with interval algebra) for online recognition, planning, and execution with multiple hypotheses tracking over weeks.

24.2 Monitoring

In an open dynamic environment, an actor cannot be confident that the predicted effects of its actions are going to occur. Acting in a blind open-loop manner is too brittle; it leads frequently to failure. An actor needs a closed-loop feedback to detect possible problems and correct its actions.

Monitoring is in charge of *(i)* detecting discrepancies between predictions and observations, *(ii)* diagnosing their causes, and *(iii)* taking first recovery actions. It ranges from the low-level surveillance of the execution platform, to the high-level reasoning on the appropriate goals for pursuing the mission. Discrepancies between predictions and observations can be caused by platform errors and failures, e.g., malfunctioning sensors or actuators or buggy commands. They can also be produced by unexpected events and environment contingencies that make the chosen plan inappropriate. Let us discuss successively these monitoring levels.

24.2.1 Platform Monitoring

The actor has to monitor its platform and adapt its actions to the functioning status of its sensory-motor capabilities.[3] Low-level monitoring may be needed even when the execution platform is solely computational. One may argue that this monitoring is a platform dependent issue, not a deliberation function. However, the actor's reasoning relies on models of the platform and its current status. In addition, acting, planning and learning techniques are very relevant for performing platform monitoring functions.

A set of monitoring techniques rely on signal filtering and parameter identification for fault detection and identification. Several methods, surveyed in [895, 227], use statistical recognition or neural net classifiers. Model-based methods usually take as input a triple *(System description, Components, Observation)* where the first term is a model of the platform, the second a finite list of its components, the third is an assertion inconsistent with the model expressing the observed fault. The diagnosis task is to find a minimum subset of components whose possible failure explains the observation. The framework of [78] formulates model-based diagnosis as a planning problem with information gathering and reasoning on change.

Livingstone is a model-based system for monitoring, diagnosis, and recovery for an earth observation spacecraft [823]. It relies on qualitative model-based diagnosis [1175]. The spacecraft is modeled as a collection of components. Each one is described by a graph whose nodes correspond to normal functioning states or to failure states, e.g., a valve is closed, open, or stuck. Edges are either nominal transition

[3]This level of monitoring is sometime referred to as fault detection, identification and recovery (FDIR).

commands or exogenous transition *failures*. The latter are labeled by transition probabilities; the former are associated with transition costs and preconditions of the commands. A node is associated with a set of finite domain constraints describing the component's properties in that state, for example, when the valve is closed, *inflow* = 0 and *outflow* = 0. The dynamics of each component is constrained such that, at any time, exactly one nominal transition is enabled but zero or more failure transitions are possible. Models of all components are compositionally assembled into a system where concurrent transitions compatible with the constraints and preconditions may take place. The entire model is compiled into a temporal propositional logic formula, which is queried through a specific solver (with a truth-maintenance and a conflict-directed best-first search). Two query modes are used: *(i) diagnosis*, which finds the most likely transitions consistent with the observation, and *(ii) recovery*, which finds the least cost commands that restore the system into a nominal state. Livingston is integrated with the spacecraft acting system. It computes a focused sequence of recovery commands that meets additional constraints specified by the acting system.

This and other similar model-based diagnosis systems are focused on monitoring the platform itself.[4] Monitoring the actor's interactions with a dynamic environment (for example, in searching for an object and bringing it to a user) requires other techniques, discussed next.

24.2.2 Action and Plan Monitoring

Monitoring the causal structure of a plan. A plan organizes actions as a sequence, a partial order, a chronicle, or a policy. The causal structure of the plan provides an important information for monitoring the progress of the plan. It says which effects of an action a are predicted to support which preconditions of an action a', constrained to come after a.

We have already discussed the causal structure of a plan in previous chapters, through the notion of causal links in a partial plan (Section 3.4), or the notion of causally supported assertions in a timeline (Definition 17.9). Let us briefly discuss its use for monitoring in the simple case of sequential plans.

Let $\pi = \langle a_1, \ldots, a_i, \ldots, a_k \rangle$ be a sequential plan for achieving a goal g. Goal regression defines the sequence of intermediate goals associated with π as:

$$\mathcal{G} = \langle g_1, \ldots, g_i, \ldots, g_{k+1} \rangle, \text{ with}$$
$$g_i = \gamma^{-1}(g_{i+1}, a_i) \text{ for } 1 \leq i \leq k, \text{ and } g_{k+1} = g.$$

In other words, action a_k can be performed in a state s and achieves g only if s supports g_k. Similarly, the subsequence $\langle a_{k-1}, a_k \rangle$ can be performed in a state s' and achieves g only if s' supports g_{k-1}. The entire plan π is applicable and achieves g only in a state that supports g_1.

$\mathcal{G}$ is easily defined from π and can be used to monitor the progress of π with the procedure Progress-Plan. This procedure searches $\mathcal{G}$ in reverse order, looking for the first g_i, which is supported by current state. It then performs action a_i. The

[4]They can be qualified as *proprioceptive monitoring* approaches.

goal is achieved when the current state supports $g_{k+1} = g$. If no intermediate goal is supported in s, then the plan π has failed.

```
Progress-Plan($\pi, g$)
    let $\pi = \langle a_1, \ldots, a_n \rangle$ ; $\mathcal{G} \leftarrow \langle g \rangle$
    for $a \leftarrow a_n$ to $a_1$ do
        $g \leftarrow \gamma^{-1}(g, a)$
        $\mathcal{G} \leftarrow g.\mathcal{G}$
    while True do
        $\xi \leftarrow$ observed current state
        if no $g \in \mathcal{G}$ is supported be $\xi$ then return failure
        else
            $i \leftarrow \max_j \{1 \le j \le k+1 \mid \xi$ supports $g_j\}$
            if $i = k+1$ then return success
            else perform action $a_i$
```

Algorithm 24.1. A simple monitoring of the progression of a plan

Note that the procedure Progress-Plan does not follow π sequentially. It "jumps" to the action closest to the goal that allow progressing toward g. It may also go back and repeat several times previously performed actions until the effects required by an intermediate goal are achieved.

Example 24.1. Consider a service robot for which a planner produces the following sequential plan: $\pi = \langle$move(door), open(door), move(table), pickup(tray), move(sink), putdown(tray, sink), pickup(medic), move(chest), putdown(medic,chest) $\rangle$. π says to open the door, assumed closed, because the robot cannot open it while holding the tray. If the robot observes when starting π, that the door is already open, Progress-Plan would skip the first two actions and proceed with the move(table). Later on, after picking up the medic if the robot observes that it gripper is empty, it would repeat the pickup action. □

The intermediate goals in the sequence $\mathcal{G}$ are not independent. They can be organized such as to reduce the computation for finding the largest i such that ξ support g_i. The corresponding data structure is a tabular representation of a causal graph called a *triangle table*. It has been proposed in Planex [358], an early monitoring and execution system associated with the Strips planner.

Progress-Plan alone is quite limited and remains at an abstract and simple level of monitoring. It has to be augmented with monitoring the commands refining the actions in π, with diagnosis of possible problems (that is, why the state observed after performing a_i does not support g_{i+1}) and the control of repeated actions on the basis of this diagnosis (for example, when does it make sense to repeat a pickup action).

Monitoring the invariants of a plan. An invariant of a state transition system is a condition that holds in every state of the system. For a planning problem (Σ, s_0, g), an invariant characterizes the set of reachable states of the problem. A state that violates the invariant cannot be reached from s_0 with the actions described in Σ. In other words, if φ is an invariant condition of (Σ, s_0, g), then for any plan π and any state $s \in \widehat{\gamma}(s_0, \pi)$, s supports φ. Going back to Example 24.1, if the robot has no action to lock or unlock the door, and if the door is initially unlocked, then door-status(door)=unlocked is an invariant of this domain. Note that the world *invariant* qualifies a particular domain model, not the world itself. Monitoring violation of invariant conditions allows detecting discrepancies with respect to that model.

Invariants of a planning problem can be synthesized automatically [596, 945]. Several authors have used invariants to speed up planning algorithms, e.g., [369]. However, at the acting level, we know that the assumption of a static environment does not hold: there can be other state transitions than those in Σ due to the actor's actions. For example, the door of Example 24.1 may become locked, this violating a plan that requires opening that door. The actor has to monitor that the current state supports the invariants relevant to its plan.

The invariants of a planning problem are often not sufficient for the purpose of monitoring. Many of the invariants entailed from (Σ, s_0, g) express syntactical dependencies between state variables, e.g., a locked door is necessarily closed; it cannot be open. Often, an actor has to monitor specific conditions that express the appropriate context in which its activity can be performed. For example, the robot has to monitor the status of its battery: if the charge level is below a threshold, than at most τ units of time are available in normal functioning before plugging at a recharge station. Such conditions cannot be deduced from the specification of (Σ, s_0, g); they have to be expressed specifically as monitoring rules.

An *extended planning problem* $(\Sigma, s_0, g, \varphi)$ specifies φ as the invariant to be monitored [374]. A plan π is a solution to the problem if every state $s \in \widehat{\gamma}(s_0, \pi)$ supports φ. This is also a requirement for acting: the actor has to monitor at acting time that every state observed while performing a plan π supports φ. A violation of this condition, due to exogenous event or malfunction, means a failure of the plan. It allows early detection of infeasible goals or actions, even if the subsequent actions in π appear to be applicable.

Elaborate versions of the preceding idea have been developed with monitoring rules in various formalisms, associated with sensing and recovery actions together with efficient incremental evaluation algorithms at acting time. For example, the approach [357] relies on the fluent calculus [977] with actions described by *normal* and *abnormal* preconditions. The former are the usual preconditions; the latter are assumed away by the planner as default; they are used as a possible explanation of a failure. For example, delivery of an object to a person may fail with abnormal preconditions of the object being lost or the person not being traceable. Abnormal effects are similarly specified. Discrepancies between expectations and observations are handled by a prioritized non-monotonic default logic and entail that default assumptions no longer hold. These explanations are ranked using relative likelihood, when available. The system can handle incomplete world models and observation updates performed while acting or on demand from the monitoring system.

An interesting variant use Linear Temporal Logic formulas to express goals as well as correctness statements and execution progress conditions [110]. A trace of the execution, observed and predicted at planning time, is incrementally checked for satisfied and violated LTL formulas. For that, a *delayed formula progression* technique evaluates at each state the set of pending formulas; it returns the set of formulas that needs to be satisfied by the remaining actions. The same technique is used both for planning (with additional precondition-effect operators and some search mechanism) and for monitoring.

Domain knowledge, expressed in description logic, enables deriving expectations of the effects of actions in a plan to be monitored during execution [168]. A first-order query language allows online matching of these expectations against observations. The parameters of actions refer to objects that have derived properties. The consistency of these properties with observations is checked. It may be undetermined, triggering observation actions. An extension handles monitoring with probabilistic models, akin to Bayesian belief update. It relies on probabilistic plans with nondeterministic actions as well as on probabilistic sensing models.

Another approach uses the *Temporal Action Logics* formalism [661] to express operators and domain knowledge [301]. Formal specifications of global constraints and dependencies, together with planning operators and control rules, are used by the planner to control and prune the search. *Monitoring formulas* are generated from the descriptive models of planning operators (preconditions, effects, and temporal constraints) and from the complete synthesized plan, for example, constraints on the persistence of causal links. This synthesis of monitoring formulas is not systematic but selective, on the basis of hand-programmed conditions of what needs to be monitored and what does not. Additional monitoring formulas are also specified by the designer in the same expressive temporal logic formalism. For example, in the application domain of [301], a UAV should have its winch retracted when its speed is above a threshold; it can exceed its continuous maximum power by a factor of η for up to τ units of time if this is followed by normal power usage for a period of at least τ'. The system produces plans with concurrent and durative actions together with conditions to be monitored during execution. These conditions are evaluated online using formula progression techniques. When actions do not achieve their desired results, or when other conditions fail, recovery via plan repair is triggered.

Integrating monitoring with operational models of actions. The previous examples of monitoring rules for a UAV express conditions on the normal functioning of the execution platform and its environment; they allow detection of deviations from the required specifications. Such a detection is naturally integrated to operational models of actions with the refinement methods of Part V. Furthermore, detections of a malfunction or a deviation may trigger events to which are associated refinement methods for taking first corrective actions specific to the context.

Refinement methods are adequate for expressing monitoring activities; RAE procedure can be used for triggering observation and commands required for monitoring. Most of the acting systems discussed in Section 14.4, such as PRS, RAP, or TCA, integrate specific methods to handle monitoring functions.

24.3 Goal Reasoning

An actor has to keep its goals in perspective to make sure that they remain feasible and relevant to its long-term mission. If needed, it has to consider alternate goals. Goal reasoning is a monitoring function at a higher level; it continuously checks for unexpected events that may interfere with current goals. A more ambitious definition considers actors that deliberate on, and self-select their objectives [16], possibly with a "motivation system" to guide their choices. The so-called "motivated" actors are assumed to be explicitly or implicitly motivated by external or internal motivations [11], the latter referring to some value functions, sometime misnamed with anthropomorphic references, e.g., to emotions or desires.

Goal reasoning has been deployed in a few experiments. An example is the DS1 spacecraft Mission Manager [823, 116], which analyses the progress of the mission and determines the goals to be satisfied for the next planning window. The selected goals are passed to the planner, together with constraints that need to be satisfied at waypoints identified by the Mission Manager (e.g., the amount of energy left in the batteries should be above a threshold at the end of the planning phase).

An analogous manager in the CPEF system [826] provides appropriate goals to the planner and controls the generation of plans. For a similar class of applications, the ARTUE system [804] detects discrepancies when executing a plan. It generates an explanation, possibly produces a new goal, and manages possible conflict between goals currently under consideration. It uses decision theory techniques to decide which goal to choose. The approach proposes an explanation system, which uses Assumption-based Truth Maintenance techniques to find the possible explanation of the observed facts. ARTUE has been extended with a facility for teaching the system new goal selection rules [914]. The goal reasoner in [1178] handles cost tradeoffs to coordinate multi-vehicle teams in AUV scenarios.

Another example is the Plan Management Agent for handling personal calendars and workflow systems [909]. It addresses the following functions:

- Commitment management: commits to a plan already produced, and avoids new plans that conflict with the existing ones.
- Alternative assessment: decides which of the possible alternative goals and plans should be kept or discarded.
- Plan control: decides when and how to generate a plan.
- Coordination with other agents: takes into account others' commitments and the cost of decisions involving their plans.

That system relies on temporal and causal reasoning. It is able to plan with partial commitments that can be further refined later.

Let us also mention a class of approaches, called *Goal Driven Autonomy* for reasoning about possibly conflicting goals and synthesizing new ones. These approaches are surveyed in [486, 1120]. The former surveys a number of architectures supporting goal reasoning in intelligent systems. The latter reviews several contributions on various techniques for goal monitoring, goal formulation, and goal management, organized within a comprehensive goal reasoning analysis framework.

Appendices

Free pre-publication, for personal use only. To be published by Cambridge University Press.

A Graphs and Search

This appendix provides background information about nondeterministic state-space search in Section A.1, and And/Or search in Section A.2.

A.1 Nondeterministic State-Space Search

Many of the planning algorithms in this book have been presented as nondeterministic search algorithms and can be described as instances of Algorithm A.1, Nondeterministic-Search. In most implementations of these algorithms, Line 3 corresponds to trying several members of R sequentially in a trial-and-error fashion. The "nondeterministically choose" command is an abstraction that lets us ignore the precise order in which those values are tried, so we can discuss properties that are shared by a wide variety of algorithms that search the same space of partial solutions, even though those algorithms may visit different nodes of that space in different orders.

```
Nondeterministic-Search($P$)                                  // Iterative version
    $\pi \leftarrow$ an initial partial solution for $P$
1   while $\pi$ is not a solution for $P$ do
2       $R \leftarrow$ {candidate refinements of $\pi$}
        if $R = \varnothing$ then return failure
3       nondeterministically choose $r \in R$
        $\pi \leftarrow$ refine($\pi, r$)
    return $\pi$

Nondeterministic-Search($P, \pi$)                             // Recursive version
1   if $\pi$ is a solution for $P$ then return $\pi$
2   $R \leftarrow$ {candidate refinements of $\pi$}
    if $R = \varnothing$ then return failure
3   nondeterministically choose $r \in R$
    $\pi \leftarrow$ refine($\pi, r$)
    return Nondeterministic-Search($P, \pi$)
```

Algorithm A.1. Iterative and recursive versions of nondeterministic search. P is a search problem, and π is an initial partial solution.

To visualize how nondeterministic search works, suppose we run it on a nondeterministic Turing machine. Let $\psi(P)$ be a process produced by calling Nondeterministic-Search on a search problem P. Whenever this process reaches Line 3, it replaces $\psi(P)$ with $|R|$ copies of $\psi(P)$ running in parallel: one copy for each $r \in R$. Each process

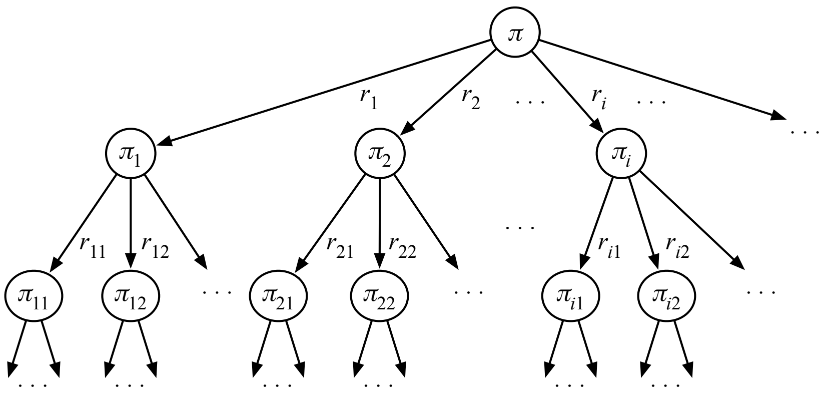


Figure A.1. Search tree for Nondeterministic-Search. Each branch represents a possible refinement.

corresponds to a different execution trace of $\psi(P)$, and each execution trace follows one of the paths in $\psi(P)$'s search tree (see Figure A.1). Each execution trace that terminates will either return failure or return a purported answer to P. Two desirable properties for ψ are *soundness* and *completeness*, which are defined as follows:

- ψ is *sound* over a set of search problems $\mathbf{P}$ if for every $P \in \mathbf{P}$ and every execution trace of $\psi(P)$, if the trace terminates and returns a value $\pi \neq$ failure, then π is a solution for P. This will happen if the solution test in Line 1 is sound.
- ψ is *complete* over $\mathbf{P}$ if for every $P \in \mathbf{P}$, if P is solvable then at least one execution trace of $\psi(P)$ will return a solution for P. This will happen if each set of candidate refinements in Line 2 are complete, that is, if it includes all of the possible refinements for π.

This model of nondeterministic search is quite similar to the one in [366, 246], which later came to be known as *angelic nondeterminism* [115, 148].

In deterministic implementations of nondeterministic search, the nondeterministic choice is replaced with a way to decide which nodes of the search tree to visit, and in what order. The simplest case is depth-first backtracking, which we can get from the recursive version of Nondeterministic-Search by making a nearly trivial modification: change the nondeterministic choice to a loop over the elements of R. For this reason, the nondeterministic choice points in nondeterministic search algorithms are sometimes called *backtracking points*.

Algorithm A.2, Deterministic-Search, is a general deterministic search algorithm. Depending on the *node-selection strategy*, that is, the technique for selecting π in line Line 1, we can get a depth-first search, breadth-first search, or a best-first search. Furthermore, by making some modifications to the pseudocode, we can get GBFS, A*, iterative deepening, branch and bound, and other kinds of search (see Chapter 3).

Earlier we said that Nondeterministic-Search is sound and complete if its solution test is sound and its sets of candidate refinements are complete (i.e., each set includes all of the possible refinements). Deterministic-Search is sound under those conditions, but whether it is complete depends on the node-selection strategy. For example, with

```
Deterministic-Search($P$)
    $\pi \leftarrow$ initial partial solution
    $\Pi \leftarrow \{\pi\}$
    while $\Pi \neq \varnothing$ do
1       select $\pi \in \Pi$
        remove $\pi$ from $\Pi$ if $\pi$ is a solution for $P$ then return $\pi$
        $R \leftarrow$ {candidate refinements for $\pi$}
        foreach $r \in R$ do
            $\pi \leftarrow$ refine($\pi, r$)
            add $\pi'$ to $\Pi$
        return failure
```

Algorithm A.2. A deterministic version of Nondeterministic-Search. Depending on how π is selected in Line 1, the algorithm can do a depth-first search, breadth-first search, or best-first search.

breadth-first node selection it will be complete, but not with depth-first node selection unless the search space is finite. Although completeness is a desirable property, other considerations can often be more important: for example, the memory requirement usually is exponentially larger for a breadth-first search than for a depth-first search.

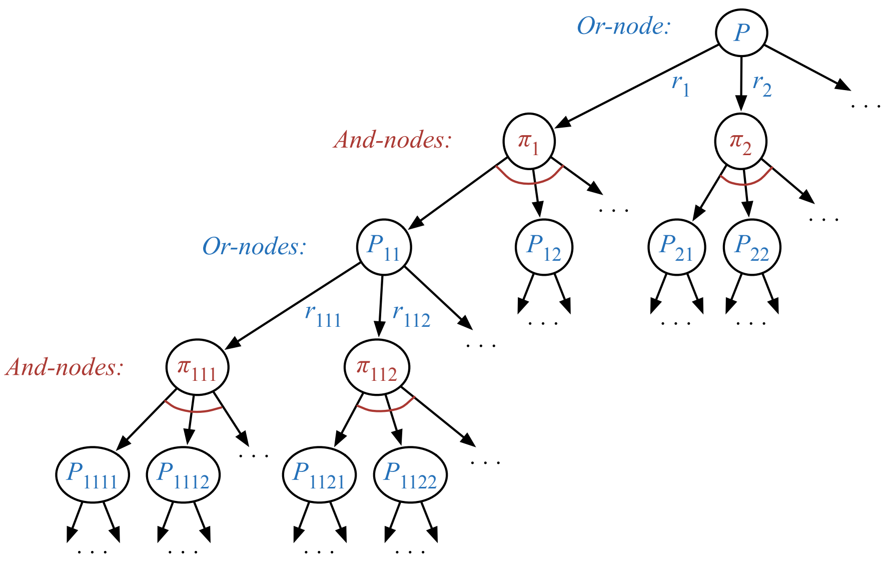


Figure A.2. Search tree for Algorithm A.3. Each Or-node represents a call to Or-Branch, and the edges below it represent members of R. Each And-node represents a call to And-Branch, and the edges below it represent subproblems of π.

A.2 And/Or Search

In addition to choosing among alternative refinements, some search algorithms involve decomposing a problem P into a set of subproblems $P_1, \ldots, P_n$ whose solutions will provide a solution for P. Such algorithms usually can be described as instances of a nondeterministic And/Or search algorithm, Algorithm A.3. The search space is an And/Or tree like the one in Figure A.2.

```
And-Or-Search($P$)
    return Or-Branch($P$)

Or-Branch($P$)
    $R \leftarrow$ {candidate refinements for $P$}
    if $R = \varnothing$ then return failure
1   nondeterministically choose $r \in R$
    $\pi \leftarrow$ refine($P, r$)
    return And-Branch($P, \pi$)

And-Branch($P, \pi$)
    if $\pi$ is a solution for $P$ then return $\pi$
    $\{P_1, \ldots, P_n\} \leftarrow$ {unsolved subproblems in $\pi$}
2   foreach $P_i \in \{P_1, \ldots, P_n\}$ do
        $\pi_i \leftarrow$ Or-Branch($P_i$)
        if $\pi_i$ = failure then return failure
    if $\pi_1, \ldots, \pi_n$ are not compatible then return failure
    incorporate $\pi_1, \ldots, \pi_n$ into $\pi$
    return $\pi$
```

Algorithm A.3. A generic nondeterministic And/Or search algorithm. P is an And/Or search problem, and π is a partial solution.

We will not present a deterministic version of And-Or-Search here because the details are somewhat complicated and generally depend on the nature of the problem domain. For example, unlike Line 1, Line 2 is not a backtracking point. The subproblems $P_1, \ldots, P_n$ must all be solved to solve P, and not every combination of solutions will be compatible. For example, if P_1 and P_2 are "find a container c and bring it to location l" and "put every book at location l into c," a solution to P_1 is useful for solving P_2 only if the container c is large enough to contain all of the books.

A.3 Strongly Connected Components of a Graph

Let $G = (V, E)$ be a directed graph. A strongly connected component of G is a subset C of V such that every vertex of C is reachable from every other vertex of C. The relation $\sim$ on vertices can be defined as follows: $v \sim v'$ iff either $v = v'$ or v is reachable from v' and v' is reachable from v. It is an equivalence relation on V. It partitions V into equivalence classes, each being a strongly connected component of G. Furthermore, the set of strongly connected components of G is a directed acyclic graph that has an edge from C to C' when there is a vertex in C' reachable from a vertex in C.

Tarjan's algorithm [1078] finds in a single depth-first traversal of G its strongly connected components. Each vertex is visited just once. Hence the traversal organizes G as a spanning forest. Some subtrees of this forest are the strongly connected components of G. During the traversal, the algorithm associates two integers to each new vertex v it meets:

- index(v): the order in which v is met in the traversal, and
- low(v) = min{index(v')|v' reachable from v}

It is shown that index(v)=low(v) if and only if v and all its successors in a traversal subtree are a strongly connected component of G.

```
Tarjan($v$)
  index($v$) $\leftarrow$low($v$) $\leftarrow i$
  $i \leftarrow i + 1$
  push($v$,$stack$)
  for all $v'$ adjacent to $v$ do
    if index($v'$) is undefined than do
      Tarjan($v'$)
      low($v$) $\leftarrow$ min{low($v$), low($v'$}
    else if $v'$ is in $stack$ then low($v$) $\leftarrow$ min{low($v$), low($v'$}
  if index($v$)=low($v$) then do
    start a new component $C \leftarrow \varnothing$
    repeat
      $w \leftarrow$ pop($stack$) ; $C \leftarrow C \cup \{w\}$
    until $w = v$
```

Algorithm A.4. Tarjan's algorithm for finding strongly connected components of a graph.

This is implemented in Algorithm A.4 as a recursive procedure with a stack mechanism. At the end of a recursion on a vertex v, if the condition index(v)=low(v) holds, then v and all the vertices above v in the stack (i.e., those below v in the depth-first traversal tree) constitute a strongly connected component of G.

With the appropriate initialization ($i \leftarrow 0$, $stack \leftarrow \varnothing$ and index undefined everywhere), Tarjan(v) is called once for every $v \in V$ such that index(v) is undefined. The algorithm run in $0(|V| + |E|)$. It finds all the strongly connected components of G in the reverse order of the topological sort of the DAG formed by the components, that is, if (C, C') is an edge of this DAG, then C' will be found before C.

B Other Mathematical Background

This appendix recapitulates the terminology, notations and a few basic definitions of algebra and calculus used in this book.

B.1 Metrics and distances

A space X is metric if it can be associated with a *metric function*, that is a function $f : X \times X \longrightarrow \mathbb{R}^+$ which is, $\forall x, x', x'' \in X$,

- reflexive: $f(x, x') = 0$ if and only if $x = x'$,
- symmetrical: $f(x, x') = f(x', x)$, and
- triangular: $f(x, x') + f(x', x'') \geq f(x, x'')$.

A metric function is also called a *distance* because it allows defining distances between points in X.

For the space $X = \mathbb{R}^n$, a point $\mathbf{x} \in X$ is a vector $[x_1, \ldots, x_n]$. X has family of metrics given by:

$$\delta_p(\mathbf{x}, \mathbf{x}') = [\sum_{1 \leq j \leq n} |x_j - x'_i|^p]^{1/p}$$

Two well-known distances in this family δ_p are the Manhattan distance δ_1, and the Euclidian distance δ_2:

$$\delta_1(\mathbf{x}, \mathbf{x}') = \sum_{1 \leq j \leq n} |x_j - x'_j)|$$

$$\delta_2(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{1 \leq j \leq n} (x_j - x'_j)^2}$$

The limit case of δ_p when p tends to infinity can also be very useful:

$$\delta_\infty(\mathbf{x}, \mathbf{x}') = \max_j\{|x_j - x'_j|\}.$$

A variant of δ_2, called the mean squared error, is: $1/n \sum_{1 \leq j \leq n} (x_j - x'_j)^2$.

B.2 Vectors and matrices

We summarize here useful notations and definitions in linear algebra.

A vector is an ordered list of elements in some scalar field, in our case a list of real values. A *column* vector is denoted as: $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n, x_i$ is the i^{th} element of $\mathbf{x}$, n is its dimension. The *transpose* operation $\top$ of a row vector $[x_1, \ldots, x_n]^\top$ is a column vector, and vice versa.

Free pre-publication, for personal use only. To be published by Cambridge University Press.

A matrix is a 2D array of $n \times m$ values:

$$\mathbf{A}_{(n,m)} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix}$$

We denote $\mathbf{A}_{(n,m)} = \left[a_{i,j}\right]$, $1 \leq i \leq n$, $1 \leq j \leq m$, when clear from the context. $\mathbf{A}^\top_{m,n}$ is the transpose of $\mathbf{A}$, it turns its rows into columns and its columns into rows.

Operations on vectors and matrices. For two vectors $\mathbf{x} = [x_1, \ldots, x_n]^\top$ and $\mathbf{y} = [y_1, \ldots, y_m]^\top$, and two matrices $\mathbf{A}$ and $\mathbf{B}$, we denote:

- $\mathbf{x} + \mathbf{y} = [x_1 + y_1, \ldots, x_n + y_n]$: the vector *sum*; $\mathbf{x} - \mathbf{y}$ is the difference ;
- $\mathbf{x} \cdot \mathbf{y}$: the *dot product* (or *inner product*); it is a scalar $\mathbf{x} \cdot \mathbf{y} = \sum_{1 \leq i \leq n} x_i y_i$;
- $\mathbf{x} \times \mathbf{y}$: the *cross product*; it is a vector $\mathbf{x} \times \mathbf{y} = [x_1 y_1, \ldots, x_n y_n]^\top$.

The sum, difference, dot and cross products are commutative operations defined only for vectors of the same dimension (i.e., $m = n$).

- $\mathbf{x} \otimes \mathbf{y}$: the *outer product*; it is a matrix $\mathbf{x} \otimes \mathbf{y} = A_{(n,m)} = [a_{i,j}]$ with $a_{i,j} = x_i y_j$.

The outer product is not commutative: $\mathbf{x} \otimes \mathbf{y} = (\mathbf{y} \otimes \mathbf{x})^\top$; it does not require $m = n$.

- $\mathbf{A} \times \mathbf{B}$: the *product* of two matrices: $\mathbf{A}_{(n,n')} \times \mathbf{B}_{(n',m)} = \mathbf{C}_{(n,m)} = \left[c_{i,j}\right]$ with $c_{i,j} = \sum_{1 \leq k \leq n'} a_{i,k} b_{k,j}$

$\mathbf{A}$ has as many columns as rows in $\mathbf{B}$ (i.e., n').

By viewing a column vector $\mathbf{y}^\top$ as a matrix of dimension $(m, 1)$, the product of matrix $\mathbf{C}_{(n,m)}$ with vector $\mathbf{y}^\top_{(m,1)}$ is a vector $\mathbf{C} \times \mathbf{y}^\top = [\sum_{1 \leq j \leq m} c_{1,j} y_j, \ldots, \sum_{1 \leq j \leq m} c_{i,j} y_j, \ldots, \sum_{1 \leq j \leq m} c_{n,j} y_j]^\top$ of dimension n. Similarly for the product $\mathbf{x}_n \times \mathbf{C}_{(n,m)} = [\sum_{1 \leq i \leq n} x_i c_{i,1}, \ldots, \sum_{1 \leq i \leq m} x_i c_{i,m}]$ is a vector of dimension m.

Applying a function $f : \mathbb{R} \to \mathbb{R}$ to a vector $\mathbf{x}$ gives the vector of f applied to the elements of $\mathbf{x}$: $f(\mathbf{x}) = [f(x_1), \ldots, f(x_n)]^\top$. Similarly for a matrix $f(\mathbf{A}) = \left[f(a_{i,j})\right]$.

The norm of a vector in $\mathbb{R}^n$ is a mapping in $\mathbb{R}^+$, with properties similar to a distance. In particular the $p-$norm is defined as:

$$\|\mathbf{x}\|_p = [\sum_{1 \leq i \leq n} |x_i|^p]^{1/p}$$

We'll also use the squared norm: $\|\mathbf{x}\|^2 = \sum_{1 \leq i \leq n} x_i^2$. Note that the Euclidian distance is $\delta_2(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|$.

Rotation matrices. Consider a reference frame $\mathcal{F}_\varphi$ attached to a polyhedra φ (Figure 20.2). $\mathcal{F}_\varphi$ is located in a global frame $\mathcal{F}$ with 6 parameters $[x_0, y_0, z_0, \alpha, \beta, \varphi]$. A point $[x_\varphi, y_\varphi, z_\varphi]$ of φ is located in $\mathcal{F}_\varphi$ with the equations defining φ. This point can be positioned in $\mathcal{F}$ with the following equation:

$$[x, y, z]^\top = \rho_\alpha \times \rho_\beta \times \rho_\varphi \times [x_\varphi, y_\varphi, z_\varphi]^\top + [x_0, y_0, z_0]^\top. \tag{B.1}$$

The three rotation matrices are:

$$\rho_\alpha = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix},\ \rho_\beta = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ \text{-}\sin\beta & 0 & \cos\beta \end{bmatrix},\ \rho_\varphi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix}.$$

B.3 Derivative and Gradient

Let h be the composition of two functions f and g, i.e., $h(x) = f(g(x))$. The derivative chain rule states that $h'(x) = f'(g(x))g'(x)$. In Leibniz notation:

$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}.$$

The rule applies for the composition of n functions: for $f(x) = f_1(f_2(\ldots(f_n(x))$

$$\frac{df}{dx} = \frac{df_1}{df_2}\cdots\frac{df_n}{dx}.$$

The gradient of a multivariable function $f(x_1, \ldots, x_n)$ is a vector of its partial derivatives, denoted $\nabla f = [\partial f/\partial x_1, \ldots, \partial f/\partial x_n]^\top$.

List of Algorithms

Free pre-publication, for personal use only. To be published by Cambridge University Press.

Bibliographic Abbreviations

AAAI	AAAI Conference on Artificial Intelligence
AAMAS	International Conference on Autonomous Agents and Multi-agent Systems
ACL	Annual Meeting of the Association for Computational Linguistics
ADPRL	IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning
AIIDE	AAAI Conference on AI and Interactive Digital Entertainment
AIJ	Artificial Intelligence (Journal)
AIMag	AI Magazine
AIPS	International Conference on AI Planning Systems
AIxIA	International Conference of the Italian Association for Artificial Intelligence
AIMA	Artificial Intelligence: A Modern Approach
AMAI	Annals of Mathematics and Artificial Intelligence
ARCRAS	Annual Review of Control, Robotics, and Autonomous Systems
arXiv	arxiv.org
ASTRA	Symposium on Advances in Space Technologies in Robotics and Automation
CACM	Communications of the Association for Computing Machinery
CASE	IEEE International Conference on Automation Science and Engineering
CAV	International Conference on Computer Aided Verification
CG	International Conference on Computers and Games
CI	Computational Intelligence
COLT	Annual Conference on Computational Learning Theory
CONCUR	International Conference on Concurrency Theory
CP	International Conference on Principles and Practice of Constraint Programming
CSUR	ACM Computing Surveys
CVPR	IEEE Conference on Computer Vision and Pattern Recognition
ECAI	European Conference on Artificial Intelligence
ECML	European Conference on Machine Learning
ECML PKDD	European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases
ECP	European Conference on Planning
ESWC	European Semantic Web Conference
ETFA	IEEE International Conference on Emerging Technologies and Factory Automation
FLAIRS	International Florida AI Research Society Conference
FOCS	Annual Symposium on Foundations of Computer Science
FOIKS	International Symposium on Foundations of Information and Knowledge Systems
GDC	Game Developers Conference
HDIP	ICAPS Workshop on Heuristics for Domain-independent Planning
HPlan	ICAPS Workshop on Hierarchical Planning
HSDIP	ICAPS Workshop on Heuristics and Search for Domain-independent Planning
Humanoids	IEEE-RAS International Conference on Humanoid Robots
ICAA	International Conference on Autonomous Agents
ICALP	International Colloquium on Automata, Languages and Programming
ICAPS	International Conference on Automated Planning and Scheduling
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
ICMLA	IEEE International Conference on Machine Learning and Applications
ICRA	IEEE International Conference on Robotics and Automation
ICTAI	IEEE International Conference on Tools with Artificial Intelligence
ICWS	IEEE International Conference on Web Services

Free pre-publication, for personal use only. To be published by Cambridge University Press.

IJCAI	International Joint Conference on Artificial Intelligence
IJCNN	International Joint Conference on Neural Networks
IJRR	International Journal of Robotics Research
ILP	International Conference on Inductive Logic Programming
IROS	IEEE/RSJ International Conference on Intelligent Robots and Systems
i-SAIRAS	International Symposium on Artificial Intelligence, Robotics and Automation in Space
ISRR	International Symposium on Robotics Research
ISWC	International Semantic Web Conference
IWPSS	International Workshop on Planning and Scheduling for Space
JAAMAS	(Journal of) Autonomous Agents and Multi-Agent Systems
JACM	Journal of the Association for Computing Machinery
JAIR	Journal of Artificial Intelligence Research
JETAI	Journal of Experimental & Theoretical Artificial Intelligence
JFR	Journal of Field Robotics
JIRS	Journal of Intelligent and Robotic Systems
JMLR	Journal of Machine Learning Research
JMAA	Journal of Mathematical Analysis and Applications
KEPS	ICAPS Workshop on Knowledge Engineering for Planning and Scheduling
KER	The Knowledge Engineering Review
KI	Annual German Conference on Artificial Intelligence (Künstliche Intelligenz)
KR	International Conference on Principles of Knowledge Representation and Reasoning
LNAI	Lecture Notes in Artificial Intelligence
LNCS	Lecture Notes in Computer Science
ML	Machine Learning
NeurIPS	Advances in Neural Information Processing Systems
PAMI	IEEE Transactions on Pattern Analysis and Machine Intelligence
PlanEx	ICAPS Workshop on Planning and Plan Execution for Real-World Systems
PLANSIG	Workshop of the UK Planning and Scheduling Special Interest Group
PNAS	Proceedings of the National Academy of Sciences of the United States of America
POPL	ACM Conference on Principles of Programming Languages
PRSA	Proceedings of the Royal Society A: Mathematics, Physics, and Engineering Sciences
RAS	Robotics and Autonomous Systems (Journal)
RSS	Robotics: Science and Systems (Conference)
SICOMP	SIAM Journal on Computing
SMC	IEEE Transactions on Systems, Man, and Cybernetics
SOCS	International Symposium on Combinatorial Search
TAC	IEEE Transactions on Automation and Control
TAC	IEEE Transactions on Automatic Control
TCIAIG	IEEE Transactions on Computational Intelligence and AI in Games
TCS	Theoretical Computer Science
TCST	IEEE Transactions on Control Systems Technology
TG	IEEE Transactions on Games
TDKE	IEEE Transactions on Knowledge and Data Engineering
TIME	International Symposium on Temporal Representation and Reasoning
T-ITS	IEEE Transactions on Intelligent Transportation Systems
TIV	IEEE Transactions on Intelligent Vehicles
TMECH	IEEE/ASME Transactions on Mechatronics
TRA	IEEE Transactions on Robotics and Automation
T-RO	IEEE Transactions on Robotics
UAI	Conference on Uncertainty in Artificial Intelligence

Bibliography

[1] M. Aarup, et al. OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In *Intelligent Scheduling*. Morgan Kaufmann, 1994.

[2] N. Ab Azar, et al. From inverse optimal control to inverse reinforcement learning: A historical review. *Annual Reviews in Control*, 2020.

[3] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *ICML*, 2004.

[4] P. Abbeel, et al. An application of reinforcement learning to aerobatic helicopter flight. In *NeurIPS*, 2006.

[5] P. Abbeel, et al. Using inaccurate models in reinforcement learning. In *ICML*, 2006.

[6] P. Abbeel, et al. Autonomous helicopter aerobatics through apprenticeship learning. *IJRR*, 2010.

[7] T. Abdul-Razaq and C. Potts. Dynamic programming state-space relaxation for single-machine scheduling. *Jour. of the Operational Research Society*, 1988.

[8] M. Abdulaziz and F. Kurz. Formally verified sat-based ai planning. In *AAAI*, 2023.

[9] J. Achiam, et al. Towards characterizing divergence in deep Q-learning. *arXiv:1903.08894*, 2019.

[10] S. Adali, et al. Representing and reasoning with temporal constraints in multimedia presentations. In *TIME*, 2000.

[11] U. Addison. Human-inspired goal reasoning implementations: A survey. *Cognitive Systems Research*, 2024.

[12] C. C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. MIT Press, 2018.

[13] J. M. Agosta. Formulation and implementation of an equipment configuration problem with the SIPE-2 generative planner. In *AAAI-95 Spring Symposium on Integrated Planning Applications*, 1995.

[14] D. J. Agravante, et al. Learning neuro-symbolic world models with conversational proprioception. In *ACL*, 2023.

[15] J. S. Aguas, et al. Synthesis of procedural models for deterministic transition systems. *arXiv:2307.14368*, 2023.

[16] D. W. Aha. Goal reasoning: Foundations, emerging applications, and prospects. *AI Magazine*, 2018.

[17] M. Ahn, et al. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv:2204.01691*, 2022.

[18] D. Aineto, et al. Learning STRIPS action models with classical planning. *AIJ*, 2019.

[19] S. Ait Bouhsain, et al. Simultaneous action and grasp feasibility prediction for task and motion planning through multi-task learning. In *IROS*, 2023.

[20] S. Ait Bouhsain, et al. Extending task and motion planning with feasibility prediction: Towards multi-robot manipulation planning of realistic objects. *IROS*, 2024.

[21] I. Akkaya, et al. Solving Rubik's cube with a robot hand. *arXiv:1910.07113*, 2019.

[22] R. Alami, et al. A geometrical approach to planning manipulation tasks. the case of discrete placements and grasps. In *ISRR*, 1989.

[23] A. Albore and P. Bertoli. Generating safe assumption-based plans for partially observable, nondeterministic domains. In *AAAI*, 2004.

[24] R. Alford, et al. Translating HTNs to PDDL: A small amount of domain knowledge can go a long way. In *IJCAI*, 2009.

[25] R. Alford, et al. Plan aggregation for strong cyclic planning in nondeterministic domains. *AIJ*, 2014.

[26] R. Alford, et al. On the feasibility of planning graph style heuristics for HTN planning. In *ICAPS*, 2014.

[27] R. Alford, et al. Tight bounds for HTN planning with task insertion. In *IJCAI*, 2015.

[28] R. Alford, et al. Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems. In *ICAPS*, 2016.

[29] J. Allen. Towards a general theory of action and time. *AIJ*, 1984.

[30] J. Allen. Temporal reasoning and planning. In J. Allen, et al., editors, *Reasoning about Plans*. Morgan Kaufmann, 1991.

[31] J. F. Allen. Maintaining knowledge about

Free pre-publication, for personal use only. To be published by Cambridge University Press.

temporal intervals. *CACM*, 1983.

[32] J. F. Allen. Planning as temporal reasoning. In *KR*, 1991.

[33] J. F. Allen and J. A. Koomen. Planning using a temporal world model. In *IJCAI*, 1983.

[34] E. Altman. *Constrained Markov Decision Processes*, volume 7. CRC Press, 1999.

[35] S. Aluru. Lagged Fibonacci random number generators for distributed memory parallel computers. *Jour. of Parallel and Distributed Computing*, 1997.

[36] J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *AAAI*, 1988.

[37] E. Amir and A. Chang. Learning partially observable deterministic action models. *JAIR*, 2008.

[38] G. Anderson, et al. Neurosymbolic reinforcement learning with formally verified exploration. *NeurIPS*, 2020.

[39] D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI*, 2002.

[40] D. Andre, et al. Generalized prioritized sweeping. In *NeurIPS*, 1997.

[41] D. Andre, et al. Generalized prioritized sweeping. *NeurIPS*, 1997.

[42] J. Andreas, et al. Modular multitask reinforcement learning with policy sketches. In *ICML*, 2017.

[43] M. Andrychowicz, et al. Hindsight experience replay. In *NeurIPS*, 2017.

[44] D. Angluin, et al. Learning regular languages via alternating automata. In *IJCAI*, 2015.

[45] B. Ans, et al. Self-refreshing memory in artificial neural networks: Learning temporal sequences without catastrophic forgetting. *Connection Science*, 2004.

[46] Anthropic AI. The claude 3 model family: Opus, sonnet, haiku, 2024. Online report.

[47] G. Antonelli, et al. Underwater robotics. In *Handbook of Robotics*. Springer, 2008.

[48] M. Araya-Lopez, et al. A closer look at MOMDPs. In *ICTAI*, 2010.

[49] S. J. Arfaee, et al. Learning heuristic functions for large state spaces. *AIJ*, 2011.

[50] B. D. Argall, et al. A survey of robot learning from demonstration. *RAS*, 2009.

[51] J. A. Arjona-Medina, et al. Rudder: Return decomposition for delayed rewards. *NeurIPS*, 2019.

[52] S. Arora and P. Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *AIJ*, 2021.

[53] K. Arulkumaran, et al. A brief survey of deep reinforcement learning. *arXiv:1708.05866*, 2017.

[54] D. Arumugam, et al. Deep reinforcement learning from policy-dependent human feedback. *arXiv:1902.04257*, 2019.

[55] D. Arumugam, et al. An information-theoretic perspective on credit assignment in reinforcement learning. *arXiv:2103.06224*, 2021.

[56] C. Arzate Cruz and T. Igarashi. A survey on interactive reinforcement learning: Design principles and open challenges. In *ACM Designing Interactive Systems Conf. (DIS)*, 2020.

[57] M. Asai. Unsupervised grounding of plannable first-order logic representation from images. In *ICAPS*, 2019.

[58] M. Asai and A. Fukunaga. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *AAAI*, 2018.

[59] K. J. Åström. Optimal control of Markov decision processes with incomplete state estimation. *JMAA*, 1965.

[60] A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *ICRA*, volume 1, 2002.

[61] A. Attia and S. Dayan. Global overview of imitation learning. *arXiv:1801.06503*, 2018.

[62] T.-C. Au and D. S. Nau. The incompleteness of planning with volatile external information. In *ECAI*, Aug. 2006.

[63] T.-C. Au, et al. On the complexity of plan adaptation by derivational analogy in a universal classical planning framework. In *European Conf. on Case-Based Reasoning (ECCBR)*, Sept. 2002.

[64] N. F. Ayan, et al. HOTRiDE: Hierarchical ordered task replanning in dynamic environments. In *PlanEx*, 2007.

[65] F. Baader, et al., editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge Univ. Press, 2003.

[66] P. BAAI. Plan4mc: Skill reinforcement learning and planning for open-world minecraft tasks. *arXiv:2303.16563*, 2023.

[67] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *AIJ*, 2000.

[68] F. Bacchus and Q. Yang. The downward refinement property. In *IJCAI*, 1991.

[69] C. Bäckström. Planning in polynomial time: The SAS-PUB class. *CI*, 1991.

[70] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. In *IJCAI*, 1993.

[71] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *CI*, 1995.

[72] E. Badouel, et al. *Petri net synthesis*. Springer, 2015.

[73] S. Badreddine and M. Spranger. Injecting prior knowledge for transfer learning into reinforcement learning algorithms using logic tensor networks. *arXiv:1906.06576*, 2019.

[74] S. Badreddine, et al. Logic tensor networks. *AIJ*, 2022.

[75] A. Bai and S. Russell. Efficient reinforcement learning with hierarchies of machines by leveraging internal transitions. In *IJCAI*, 2017.

[76] T. Bai, et al. Temporal graph neural networks for social recommendation. In *IEEE Internat. Conf. on Big Data*, 2020.

[77] J. A. Baier, et al. A heuristic search approach to planning with temporally extended preferences. *AIJ*, 2009.

[78] J. A. Baier, et al. Diagnostic problem solving: a planning perspective. In *KR*, 2014.

[79] M. Ball and R. C. Holte. The compression power of symbolic pattern databases. In *ICAPS*, 2008.

[80] Y. Bansod, et al. HTN replanning from the middle. In *FLAIRS*, May 2022.

[81] P. Baptiste, et al. Constraint-based scheduling and planning. In F. Rossi, et al., editors, *Handbook of Constraint Programming*, chapter 22. Elsevier, 2006.

[82] M. Barbier, et al. Implementation and flight testing of an onboard architecture for mission supervision. In *Internat. Unmanned Air Vehicle Systems Conf.*, 2006.

[83] J. Barraquand, et al. Numerical potential field techniques for robot path planning. *SMC*, 1992.

[84] J. Barraquand, et al. A random sampling scheme for path planning. *IJRR*, 1997.

[85] A. Barrett and D. S. Weld. Characterizing subgoal interactions for planning. In *IJCAI*, 1993.

[86] A. Barrett and D. S. Weld. Partial order planning: Evaluating possible efficiency gains. *AIJ*, 1994.

[87] A. Barrett, et al. UCPOP user's manual. Technical Report TR-93-09-06, Univ. of Washington, 1993.

[88] C. Barrett, et al. The SMT-LIB standard: Version 2.0. In *International Workshop on Satisfiability Modulo Theories*, 2010.

[89] S. Barrett, et al. Transfer learning for reinforcement learning on a physical robot. In *AAMAS Adaptive Learning Agents Workshop*, 2010.

[90] J. Barry, et al. DetH*: Approximate hierarchical solution of large Markov decision processes. In *IJCAI*, 2011.

[91] R. Barták and D. Toropila. Reformulating constraint models for classical planning. In *FLAIRS*, 2008.

[92] R. Barták and D. Toropila. Enhancing constraint models for planning problems. In *FLAIRS*, 2009.

[93] R. Barták, et al. *An Introduction to Constraint-Based Temporal Reasoning*. Morgan&Claypool, 2014.

[94] R. Barták, et al. Validation of hierarchical plans via parsing of attribute grammars. In *ICAPS*, 2018.

[95] R. Barták, et al. A novel parsing-based approach for verification of hierarchical plans. In *ICTAI*, 2020.

[96] R. Barták, et al. Correcting hierarchical plans by action deletion. In *KR*, 2021.

[97] A. Barto and M. Duff. Monte carlo matrix inversion and reinforcement learning. *NeurIPS*, 1993.

[98] A. G. Barto, et al. Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 1981.

[99] A. G. Barto, et al. Learning to act using real-time dynamic-programming. *AIJ*, 1995.

[100] K. Bauters, et al. CAN (PLAN)+: extending the operational semantics of the BDI architecture to deal with uncertain information. In *UAI*, 2014.

[101] K. Bauters, et al. Anytime algorithms for solving possibilistic MDPs and hybrid MDPs. In *FoIKS*, 2016.

[102] M. Beetz. Structured reactive controllers: Controlling robots that perform everyday activity. In *ICAA*, 1999.

[103] M. Beetz and D. McDermott. Declarative goals in reactive plans. In *AIPS*, 1992.

[104] M. Beetz and D. McDermott. Improving robot plans during their execution. In *AIPS*, 1994.

[105] G. Behnke. *Hierarchical Planning through Propositional Logic*. PhD thesis, Ulm University, 2019.

[106] G. Behnke, et al. This is a solution! (...

but is it though?): Verifying solutions of hierarchical planning problems. In *ICAPS*, 2017.

[107] G. Behnke, et al. totSAT – totally-ordered hierarchical planning through SAT. In *AAAI*, 2018.

[108] G. Behnke, et al. Finding optimal solutions in HTN planning-a SAT-based approach. In *IJCAI*, 2019.

[109] R. Bellman. *Dynamic Programming*. Princeton Univ. Press, 1957.

[110] K. Ben Lamine and F. Kabanza. Reasoning about robot actions: a model checking approach. In M. Beetz, et al., editors, *Advances in Plan-Based Control of Robotic Agents*. 2002.

[111] Y. Bengio, et al. A neural probabilistic language model. *JMLR*, 2003.

[112] Y. Bengio, et al. Curriculum learning. In *ICML*, 2009.

[113] P. Bercher, et al. Hybrid planning heuristics based on task decomposition graphs. In *SOCS*, 2014.

[114] P. Bercher, et al. An admissible HTN planning heuristic. In *IJCAI*, 2017.

[115] R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and nondeterministic programs. *TCS*, 1986.

[116] D. Bernard, et al. Remote agent experiment DS1 technology validation report. Technical report, NASA, 2000.

[117] L. Bernardinello and L. Petrucci, editors. *PETRI NETS: 43rd Internat. Conf. on Applications and Theory of Petri Nets and Concurrency*, 2022. Springer.

[118] S. Bernardini and D. Smith. Finding mutual exclusion invariants in temporal planning domains. In *IWPSS*, 2011.

[119] S. Bernardini and D. E. Smith. Developing domain-independent search control for Europa2. In *HDIP*, 2007.

[120] S. Bernardini and D. E. Smith. Automatically generated heuristic guidance for Europa2. In *i-SAIRAS*, 2008.

[121] S. Bernardini and D. E. Smith. Towards search control via dependency graphs in Europa2. In *HDIP*, 2009.

[122] C. Berner, et al. Dota 2 with large scale deep reinforcement learning. *arXiv:1912.06680*, 2019.

[123] B. Berthomieu, et al. The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *Internat. Jour. of Production Research*, 2004.

[124] P. Bertoli, et al. MBP: a model based planner. In *IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[125] P. Bertoli, et al. Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI*, 2001.

[126] P. Bertoli, et al. A framework for planning with extended goals under partial observability. In *ICAPS*, 2003.

[127] P. Bertoli, et al. Interleaving execution and planning for nondeterministic, partially observable domains. In *ECAI*, 2004.

[128] P. Bertoli, et al. Strong planning under partial observability. *AIJ*, 2006.

[129] P. Bertoli, et al. Automated composition of Web services via planning in asynchronous domains. *AIJ*, 2010.

[130] D. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2001.

[131] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[132] D. P. Bertsekas and J. N. Tsitsiklis. An analysis of stochastics shortest path problems. *Mathematics of Operations Research*, 1991.

[133] C. Betz and M. Helmert. Planning with h^+ in theory and practice. In *KI*, 2009.

[134] R. Beutner and B. Finkbeiner. Nondeterministic planning for hyperproperty verification. In *ICAPS*, 2024.

[135] J. Bhandari and D. Russo. Global optimality guarantees for policy gradient methods. *arXiv:1906.01786*, 2019.

[136] S. Bhatnagar, et al. Incremental natural actor-critic algorithms. *NeurIPS*, 2007.

[137] F. Bianchi, et al. Monte Carlo Tree Search Planning for continuous action and state space. In *Italian Workshop on AI Robotics*. 2022.

[138] R. A. Bianchi, et al. Heuristic selection of actions in multiagent reinforcement learning. In *IJCAI*, 2007.

[139] R. A. Bianchi, et al. Accelerating autonomous learning by using heuristic selection of actions. *Jour. of Heuristics*, 2008.

[140] R. A. Bianchi, et al. Transferring knowledge as heuristics in reinforcement learning: A case-based approach. *AIJ*, 2015.

[141] J. Bidot, et al. Plan repair in hybrid planning. In *KI*, 2008.

[142] J. Bidot, et al. Geometric backtracking for combined task and motion planning in

robotic systems. *AIJ*, 2015.

[143] A. Bit-Monnot, et al. FAPE: a Constraint-based Planner for Generative and Hierarchical Temporal Planning. *arXiv:2010.13121*, 2020.

[144] S. Biundo and B. Schattenberg. From abstract crisis to concrete relief – A preliminary report on combining state abstraction and HTN planning. In *ECP*, 2001.

[145] A. Blum and J. Langford. Probabilistic planning in the graphplan framework. In *ECP*, 1999.

[146] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *AIJ*, 1997.

[147] M. Boddy and T. Dean. Solving time-dependent planning problems. In *IJCAI*, 1989.

[148] R. Bodik, et al. Programming with angelic nondeterminism. In *POPL*, 2010.

[149] A. Boeing and T. Bräunl. Evaluation of real-time physics simulation systems. In *Internat. Conf. on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, 2007.

[150] J. Bohren, et al. Towards autonomous robotic butlers: Lessons learned with the PR2. In *ICRA*, 2011.

[151] R. Bommasani, et al. On the opportunities and risks of foundation models. *arXiv:2108.07258*, 2022.

[152] L. Bonassi, et al. FOND planning for pure-past linear temporal logic goals. In *ECAI*, 2023.

[153] L. Bonassi, et al. Planning for temporally extended goals in pure-past linear temporal logic. In *ICAPS*, 2023.

[154] B. Bonet. On the speed of convergence of value iteration on stochastic shortest-path problems. *Mathematics of Operations Research*, 2007.

[155] B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *AIPS*, 2000.

[156] B. Bonet and H. Geffner. Planning as heuristic search. *AIJ*, 2001.

[157] B. Bonet and H. Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *IJCAI*, 2003.

[158] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS*, 2003.

[159] B. Bonet and H. Geffner. mGPT: A probabilistic planner based on heuristic search. *JAIR*, 2005.

[160] B. Bonet and H. Geffner. Learning in depth-first search: A unified approach to heuristic search in deterministic, non-deterministic, probabilistic, and game tree settings. In *ICAPS*, 2006.

[161] B. Bonet and H. Geffner. Solving POMDPs: RTDP-Bel vs. point-based algorithms. In *IJCAI*, 2009.

[162] B. Bonet and H. Geffner. Action selection for MDPs: Anytime AO* versus UCT. In *AAAI*, 2012.

[163] B. Bonet and H. Geffner. Belief tracking for planning with sensing: Width, complexity and approximations. *JAIR*, 2014.

[164] B. Bonet and H. Geffner. Learning first-order symbolic planning representations from plain graphs. *arXiv:1909.05546*, 2019.

[165] B. Bonet and M. Helmert. Strengthening landmark heuristics via hitting sets. In *ECAI*, 2010.

[166] B. Bonet, et al. Directed unfolding of petri nets. *Trans. Petri Nets Other Model. Concurr.*, 2008.

[167] C. Borst, et al. Rollin' Justin - Mobile platform with variable base. In *ICRA*, 2009.

[168] A. Bouguerra, et al. Semantic knowledge-based execution monitoring for mobile robots. In *ICRA*, 2007.

[169] C. Boutilier, et al. Structured reachability analysis for Markov decision processes. In *UAI*, 1998.

[170] C. Boutilier, et al. Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR*, May 1999.

[171] C. Boutilier, et al. Stochastic dynamic programming with factored representations. *AIJ*, 2000.

[172] J. Boyan and M. Littman. Exact solutions to time-dependent mdps. *NeurIPS*, 2000.

[173] J. A. Boyan and M. L. Littman. Exact solutions to time dependent MDPs. In *NeurIPS*, 2001.

[174] J. A. Boyan and A. W. Moore. Learning evaluation functions to improve optimization by local search. *JMLR*, 2000.

[175] R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. In *ICAPS*, 2004.

[176] B. Braunschweig and M. Ghallab, editors. *Reflections on Artificial Intelligence for Humanity*. Springer, 2021.

[177] M. Brenner and B. Nebel. Continual planning and acting in dynamic multiagent environments. *JAAMAS*, 2009.

[178] R. Bridson. *Fluid Simulation for Computer Graphics*. CRC Press, 2015.

[179] M. A. A. Brohan, et al. Do as I can, not as I say: Grounding language in robotic affordances. *arXiv:2204.01691*, 2022.

[180] R. Brooks and T. Lozano-Pérez. A subdivision algorithm in configuration space for findpath with rotation. In *IJCAI*, 1983.

[181] N. Brown and T. Sandholm. Superhuman ai for multiplayer poker. *Science*, 2019.

[182] C. B. Browne, et al. A survey of Monte Carlo tree search methods. *TCIAIG*, 2012.

[183] V. Brusoni, et al. A spectrum of definitions for temporal model-based diagnosis. *AIJ*, 1998.

[184] V. Brusoni, et al. Qualitative and quantitative temporal constraints and relational databases: Theory, architecture, and applications. *TDKE*, 1999.

[185] V. Bruyère, et al. Active learning of mealy machines with timers. *arXiv:2403.02019*, 2024.

[186] T. Brys, et al. Reinforcement learning from demonstration through shaping. In *IJCAI*, 2015.

[187] S. Bubeck, et al. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv:2303.12712*, 2023.

[188] A. Bucchiarone, et al. Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*, 2013.

[189] C. Büchner, et al. Exploiting cyclic dependencies in landmark heuristics. In *ICAPS*, volume 31, 2021.

[190] O. Buffet and O. Sigaud, editors. *Markov Decision Processes in Artificial Intelligence*. Wiley, 2010.

[191] M. Buro. From simple features to sophisticated evaluation functions. In *CG*, 1998.

[192] L. Busoniu, et al. Optimistic planning for sparsely stochastic systems. In *ADPRL*, 2011.

[193] T. Bylander. Complexity results for extended planning. In *AAAI*, 1992.

[194] T. Bylander. The computational complexity of propositional STRIPS planning. *AIJ*, 1994.

[195] S. Caldera, et al. Review of deep learning methods in robotic grasp detection. *Multimodal Technologies and Interaction*, 2018.

[196] A. Camacho, et al. Finite LTL synthesis with environment assumptions and quality measures. In *KR*, 2018.

[197] S. Cambon, et al. A hybrid approach to intricate motion, manipulation and task planning. *IJRR*, Jan. 2009.

[198] T. Campari, et al. Online learning of reusable abstract models for object goal navigation. In *CVPR*, 2022.

[199] J. Canny. *The complexity of robot motion planning*. MIT press, 1988.

[200] Y. Cao and C. Lee. Robot behavior-tree-based task generation with large language models. *arXiv:2302.12927*, 2023.

[201] Z. Cao, et al. Temporal video-language alignment network for reward shaping in reinforcement learning. *arXiv:2302.03954*, 2023.

[202] A. Capitanelli and F. Mastrogiovanni. A framework to generate neurosymbolic PDDL-compliant planners. *arXiv:2303.00438*, 2023.

[203] J. Carbonell, et al. PRODIGY : An integrated architecture for planning and learning. In K. van Lehn, editor, *Architectures for Intelligence*. Lawrence Erlbaum Associates, 1990.

[204] J. Cardoso and R. Valette. *Petri nets*. Open Science, DOI:10.34849/zkrr-sn28, 2024.

[205] T. Carta, et al. Grounding large language models in interactive environments with online reinforcement learning. In *ICML*, 2023.

[206] L. Castano and H. Xu. Safe decision making for risk mitigation of uas. In *Internat. Conf. on Unmanned Aircraft Systems*, 2019.

[207] C. Castellini, et al. Improvements to SAT-based conformant planning. In *ECP*, 2001.

[208] C. Castellini, et al. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *AIJ*, 2003.

[209] L. Castillo, et al. Efficiently handling temporal knowledge in an HTN planner. In *ICAPS*, 2006.

[210] L. Castillo, et al. Automatic generation of temporal planning domains for e-learning problems. *Journal of Scheduling*, 2010.

[211] M. Certicky. Real-time action model learning with online algorithm 3SG. *Applied Artificial Intelligence*, 2014.

[212] A. Cesta and A. Oddi. Gaining efficiency and flexibility in the simple temporal problem. In *TIME*, 1996.

[213] A. Cesta, et al. A constraint-based method for project scheduling with time windows. *Jour. of Heuristics*, 2002.

[214] A. Champandard, et al. The AI for Killzone 2's multiplayer bots. In *GDC*, 2009.

[215] X. Chang, et al. A comprehensive survey of scene graphs: Generation and application. *PAMI*, 2021.

[216] H. J. Charlesworth and G. Montana. Solving challenging dexterous manipulation tasks with trajectory optimisation and reinforcement learning. In *ML*, 2021.

[217] E. Charniak. *Introduction to Deep Learning*. MIT Press, 2018.

[218] G. M. J. Chaslot, et al. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 2008.

[219] R. Chatilla, et al. Integrated planning and execution control of autonomous robot actions. In *ICRA*, 1992.

[220] F. Chaumette and S. Hutchinson. Visual servoing and visual tracking. In *Handbook of Robotics*. Springer, 2008.

[221] D. Chen and P. Bercher. Fully observable nondeterministic HTN planning–formalisation and complexity results. In *ICAPS*, 2021.

[222] D. Z. Chen, et al. Learning domain-independent heuristics for grounded and lifted planning. In *AAAI*, 2024.

[223] J. Chen, et al. Benchmarking large language models in retrieval-augmented generation. In *AAAI*, 2024.

[224] L. Chen, et al. Decision transformer: Reinforcement learning via sequence modeling. *NeurIPS*, 2021.

[225] P. C. Chen and Y. K. Hwang. Sandros: a dynamic graph search algorithm for motion planning. In *ICRA*, 1998.

[226] Y. Chen, et al. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. *arXiv:2306.06531*, 2023.

[227] Z. Chen, et al. Graph neural network-based fault diagnosis: a review. *arXiv:2111.08185*, 2021.

[228] C.-A. Cheng, et al. Heuristic-guided reinforcement learning. *NeurIPS*, 2021.

[229] M. Chignoli, et al. The MIT humanoid robot: Design, motion planning, and control for acrobatic behaviors. In *Humanoids*, 2021.

[230] R. Chitnis, et al. Guided search for task and motion plans using learned heuristics. In *ICRA*, 2016.

[231] H. Choset, et al. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

[232] B. Christian. *The alignment problem: How can machines learn human values?* Atlantic Books, 2021.

[233] P. F. Christiano, et al. Deep reinforcement learning from human preferences. *NeurIPS*, 2017.

[234] A. Cimatti, et al. A provably correct embedded verifier for the certification of safety critical software. In *CAV*, 1997.

[235] A. Cimatti, et al. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, 1998.

[236] A. Cimatti, et al. Strong planning in non-deterministic domains via model checking. In *AIPS*, June 1998.

[237] A. Cimatti, et al. Weak, strong, and strong cyclic planning via symbolic model checking. *AIJ*, 2003.

[238] A. Cimatti, et al. Solving temporal problems using SMT: Strong controllability. In *CP*, 2012.

[239] A. Cimatti, et al. Solving temporal problems using SMT: Weak controllability. In *AAAI*, 2012.

[240] A. Cimatti, et al. Strong temporal planning with uncontrollable durations: A state-space approach. In *AAAI*, Jan. 2015.

[241] J. Claßen, et al. Platas—integrating planning and the action language Golog. *KI*, 2012.

[242] L. Claussmann, et al. A review of motion planning for highway autonomous driving. *T-ITS*, 2019.

[243] J. D. Co-Reyes, et al. Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. In *ICML*, 2018.

[244] A. Coates, et al. Apprenticeship learning for helicopter control. *CACM*, 2009.

[245] L. C. Cobo, et al. Abstraction from demonstration for efficient reinforcement learning in high-dimensional domains. *AIJ*, 2014.

[246] J. Cohen. Non-deterministic algorithms. *ACM Computing Surveys (CSUR)*, 1979.

[247] A. Coles and A. Smith. Marvin: A heuristic search planner with online macro-action learning. *JAIR*, 2007.

[248] A. I. Coles, et al. Planning with problems requiring temporal coordination. In *AAAI*, 2008.

[249] A. J. Coles, et al. COLIN: planning with continuous linear numeric change. *JAIR*, 2012.

[250] M. Colledanchise and L. Natale. Improving the parallel execution of behavior trees.

In *IROS*, 2018.

[251] M. Colledanchise and L. Natale. Handling concurrency in behavior trees. *T-RO*, 2022.

[252] M. Colledanchise and P. Ögren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018.

[253] M. Colledanchise, et al. Learning of behavior trees for autonomous agents. *TG*, 2019.

[254] M. Colledanchise, et al. Formalizing the execution context of behavior trees for runtime verification of deliberative policies. In *IROS*. 2021.

[255] P. Conrad, et al. Flexible execution of plans with choice. In *ICAPS*, 2009.

[256] S. Coradeschi and A. Saffiotti. Perceptual anchoring: a key concept for plan execution in embedded systems. In *Advances in Plan-Based Control of Robotic Agents*. Springer-Verlag, 2002.

[257] P. I. Corke, et al. Mining robotics. In *Handbook of Robotics*. Springer, 2008.

[258] R. C. Corrêa, et al. Insertion and sorting in a sequence of numbers minimizing the maximum sum of a contiguous subsequence. *Jour. of Discrete Algorithms*, 2013.

[259] A. Couetoux. *Monte Carlo tree search for continuous and stochastic sequential decision making problems*. PhD thesis, Université Paris Sud, 2013.

[260] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In *CG*. 2006.

[261] D. Cram, et al. A complete chronicle discovery approach: application to activity analysis. *Expert Systems*, 2012.

[262] S. Cresswell, et al. Acquiring planning domain models using *LOCM*. *KER*, 2013.

[263] J. C. Culberson and J. Schaeffer. Pattern databases. *CI*, 1998.

[264] K. Currie and A. Tate. O-Plan: the open planning architecture. *AIJ*, 1991.

[265] K. Currie and A. Tate. O-Plan: The open planning architecture. *AIJ*, 1991.

[266] W. Cushing and S. Kambhampati. Replanning: A new perspective. *ICAPS*, 2005. Poster.

[267] G. Dagan, et al. Dynamic planning with a LLM. *arXiv:2308.06391*, 2023.

[268] P. Dai and E. A. Hansen. Prioritizing Bellman backups without a priority queue. In *ICAPS*, 2007.

[269] U. Dal Lago, et al. Planning with a language for extended goals. In *AAAI*, 2002.

[270] M. Dalal, et al. Imitating task and motion planning with visuomotor transformers. *arXiv:2305.16309*, 2023.

[271] M. Daniele, et al. Strong cyclic planning revisited. In *ECP*, Sept. 1999.

[272] N. T. Dantam, et al. An incremental constraint-based framework for task and motion planning. *IJRR*, 2018.

[273] I. Dasgupta, et al. Collaborating with language models for embodied reasoning. *arXiv:2302.00763*, 2023.

[274] G. De Giacomo, et al. Automatic behavior composition synthesis. *AIJ*, 2013.

[275] L. de Silva. HTN acting: A formalism and an algorithm. In *AAMAS*, 2018.

[276] L. De Silva and L. Padgham. A comparison of BDI based real-time reasoning and HTN based planning. In *Australian Joint Conf. on AI*, 2004.

[277] L. de Silva, et al. The HATP hierarchical planner: Formalisation and an initial study of its usability and practicality. In *IROS*, 2015.

[278] L. D. de Silva, et al. Towards combining HTN planning and geometric task planning. In *RSS Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications*, 2013.

[279] P. E. U. de Souza, et al. Momdp-based target search mission taking into account the human operator's cognitive state. In *ICTAI*, 2015.

[280] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *CI*, 1989.

[281] T. Dean and S.-H. Lin. Decomposition techniques for planning in stochastic domains. In *IJCAI*, 1995.

[282] T. Dean and D. McDermott. Temporal data base management. *AIJ*, 1987.

[283] T. Dean, et al. Hierarchical planning involving deadlines, travel time and resources. *CI*, 1988.

[284] T. Dean, et al. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *UAI*, 1997.

[285] T. L. Dean and M. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[286] R. Dechter, et al. Temporal constraint networks. *AIJ*, 1991.

[287] T. Degris, et al. Model-free reinforcement learning with continuous action in practice. In *American Control Conf.*, 2012.

[288] M. P. Deisenroth, et al. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2013.

[289] P. Del Moral. Nonlinear filtering: Interacting particle resolution. *Comptes Rendus de l'Académie des Sciences-Series I-Mathematics*, 1997.

[290] J.-A. Delamer, et al. Safe path planning for UAV urban operation under gnss signal occlusion risk. *RAS*, 2021.

[291] F. den Hengst, et al. Planning for potential: efficient safe reinforcement learning. *ML*, 2022.

[292] F. den Hengst, et al. Reinforcement learning with option machines. In L. D. Raedt, editor, *IJCAI*, 2022.

[293] F. D'Epenoux. A probabilistic production and inventory problem. *Management Science*, 1963.

[294] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, 1970.

[295] O. Despouys and F. Ingrand. Propice-Plan: Toward a unified framework for planning and execution. In *ECP*, 1999.

[296] M. Diaz, editor. *Petri Nets: Fundamental Models, Verification and Applications*. Wiley, 2009.

[297] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR*, 2000.

[298] M. B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *AIJ*, 2001.

[299] M. B. Do and S. Kambhampati. Sapa: A domain independent heuristic metric temporal planner. In *ECP*, 2001.

[300] P. Doherty and J. Kvarnström. TALplanner: A temporal logic based planner. *AIMag*, 2001.

[301] P. Doherty, et al. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *JAAMAS*, 2009.

[302] B. R. Donald and P. Xavier. Provably good approximation algorithms for optimal kinodynamic planning (1 & 2). *Algorithmica*, 1995.

[303] H. Dong, et al. Neural logic machines. *arXiv:1904.11694*, 2019.

[304] J. E. Doran and D. Michie. Experiments with the graph traverser program. *PRSA*, 1966.

[305] R. C. Dorf and R. H. Bishop. *Modern Control Systems*. Prentice Hall, 2010.

[306] C. Dornhege, et al. Semantic attachments for domain-independent planning systems. In *ICAPS*, 2009.

[307] C. Dousson and P. Le Maigat. Chronicle recognition improvement using temporal focusing and hierarchization. In *IJCAI*, 2007.

[308] C. Dousson, et al. Situation recognition: Representation and algorithms. In *IJCAI*, 1993.

[309] T. Drakengren and P. Jonsson. Eight maximal tractable subclasses of Allen's algebra with metric time. *JAIR*, 1997.

[310] D. Driess, et al. Deep visual heuristics: Learning feasibility of mixed-integer programs for manipulation planning. In *ICRA*, 2020.

[311] D. Driess, et al. Palm-e: An embodied multimodal language model. *arXiv:2303.03378*, 2023.

[312] N. Drougard, et al. Qualitative possibilistic mixed-observable MDPs. *arXiv:1309.6826*, 2013.

[313] Y. Du, et al. Guiding pretraining in reinforcement learning with large language models. In *ICML*, 2023.

[314] H. Duan, et al. Sim-to-real learning of footstep-constrained bipedal dynamic walking. *arXiv:2203.07589*, 2022.

[315] S. Dutta, et al. Frugal lms trained to invoke symbolic solvers achieve parameter-efficient arithmetic reasoning. In *AAAI*, 2024.

[316] F. Dvorak, et al. A flexible ANML actor and planner in robotics. In *ICAPS Wksp. on Planning and Robotics*, 2014.

[317] J. H. Eaton and L. A. Zadeh. Optimal pursuit strategies in discrete state probabilistic systems. *Transactions of the ASME*, 1962.

[318] A. Ecoffet, et al. Go-explore: a new approach for hard-exploration problems. *arXiv*, 2021.

[319] S. Edelkamp. Planning with pattern databases. In *ECP*, 2001.

[320] S. Edelkamp. Symbolic pattern databases in heuristic search planning. In *AIPS*, 2002.

[321] S. Edelkamp. Taming numbers and durations in the model checking integrated planning system. *JAIR*, 2003.

[322] R. Effinger, et al. Dynamic execution of temporally and spatially flexible reactive programs. In *AAAI Workshop on Bridging the Gap between Task and Motion Planning*, 2010.

[323] A. El-Kholy and B. Richard. Temporal and resource reasoning in planning: the ParcPlan approach. In *ECAI*, 1996.

[324] M. Elkawkagy, et al. Improving hierarchical planning performance by the use of landmarks. In *AAAI*, volume 26, 2021.

[325] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.

[326] E. Erdem, et al. Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intelligent Service Robotics*, Oct. 2012.

[327] K. Erol, et al. Semantics for hierarchical task-network planning. Technical Report CS TR-3239, Univ. of Maryland, 1994.

[328] K. Erol, et al. UMCP: A sound and complete procedure for hierarchical task-network planning. In *AIPS*, June 1994.

[329] K. Erol, et al. Complexity, decidability and undecidability results for domain-independent planning. *AIJ*, 1995.

[330] K. Erol, et al. Complexity results for HTN planning. *AMAI*, 1996.

[331] T. A. Estlin and R. J. Mooney. Learning to improve both efficiency and quality of planning. In *IJCAI*, 1997.

[332] T. A. Estlin, et al. An argument for a hybrid HTN/operator-based approach to planning. In *ECP*, 1997.

[333] C. Estrada, et al. Hierarchical SLAM: Real-time accurate mapping of large environments. *TRA*, 2005.

[334] O. Etzioni, et al. An approach to planning with incomplete information. In *KR*, 1992.

[335] EU High-Level Expert Group on AI. Ethics Guidelines for Trustworthy AI, 2019.

[336] P. Eyerich, et al. Using the context-enhanced additive heuristic for temporal and numeric planning. In *ICAPS*, 2009.

[337] R. Fakoor, et al. Meta-Q-learning. *arXiv:1910.00125*, 2019.

[338] J. Fan, et al. A theoretical analysis of deep Q-learning. *arXiv:1901.00137*, 2020.

[339] Y. Fan, et al. Heterogeneous temporal graph neural network. In *SIAM Internat. Conf. on Data Mining (SDM)*, 2022.

[340] R. Farahbod, et al. Specification and validation of the business process execution language for web services. In *Internat. Workshop on Abstract State Machines*, 2004.

[341] H. Fargier, et al. Using temporal constraint networks to manage temporal scenario of multimedia documents. In *ECAI Workshop on Spatial and Temporal Reasoning*, 1998.

[342] F. Faure, et al. Sofa: A multi-model framework for interactive physical simulation. In *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*. Springer, 2012.

[343] Z. Feng and E. A. Hansen. Symbolic heuristic search for factored Markov decision processes. In *AAAI*, 2002.

[344] Z. Feng, et al. Symbolic generalization for on-line planning. In *UAI*, 2002.

[345] Z. Feng, et al. Dynamic programming for structured continuous Markov decision problems. In *AAAI*, 2004.

[346] P. Ferber, et al. Neural network heuristics for classical planning: A study of hyperparameter space. In *ECAI*, 2020.

[347] P. Ferber, et al. Neural network heuristic functions for classical planning: Bootstrapping and comparison to other methods. In *ICAPS*, 2022.

[348] D. Ferguson, et al. Motion planning in urban environments. *JFR*, 2008.

[349] D. I. Ferguson and A. Stentz. Focussed propagation of MDPs for path planning. In *ICTAI*, 2004.

[350] F. Fernández and M. M. Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *AAMAS*, 2006.

[351] E. Feron and E. N. Johnson. Aerial robotics. In *Handbook of Robotics*. Springer, 2008.

[352] P. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *AAAI*, 2000.

[353] A. Ferrein and G. Lakemeyer. Logic-based robot control in highly dynamic domains. *RAS*, 2008.

[354] J. Ferrer-Mestres, et al. Combined task and motion planning as classical AI planning. *arXiv:1706.06927*, 2017.

[355] J. Ferret, et al. Credit assignment as a proxy for transfer in reinforcement learning. *arXiv:1907.08027*, 2019.

[356] D. A. Ferrucci, et al. Building Watson: An Overview of the DeepQA Project. *AI Mag.*, 2010.

[357] M. Fichtner, et al. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 2003.

[358] R. E. Fikes. Monitored execution of robot plans produced by STRIPS. In *IFIP*

Congress, 1971.

[359] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ*, 1971.

[360] R. E. Fikes, et al. Learning and executing generalized robot plans. *AIJ*, 1972.

[361] C. Finn, et al. Guided cost learning: Deep inverse optimal control via policy optimization. In *ML*, 2016.

[362] A. Finzi, et al. Open world planning in the situation calculus. In *AAAI*, 2000.

[363] R. J. Firby. An investigation into reactive planning in complex domains. In *AAAI*, 1987.

[364] M. Fisher, et al., editors. *Handbook of Temporal Reasoning in Artificial Intelligence*. Elsevier, 2005.

[365] G. Flórez-Puga, et al. Query-enabled behavior trees. *TCIAIG*, 2009.

[366] R. W. Floyd. Nondeterministic algorithms. *JACM*, 1967.

[367] A. Foka and P. Trahanias. Real-time hierarchical POMDPs for autonomous robot navigation. *RAS*, 2007.

[368] J. P. Forestier and P. Varaiya. Multilayer control of large Markov chains. *TAC*, 1978.

[369] M. Fox and D. Long. Utilizing automatically inferred invariants in graph construction and search. In *ICAPS*, 2000.

[370] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 2003.

[371] M. Fox, et al. Plan stability: Replanning versus plan repair. In *ICAPS*, 2006.

[372] J. Frank and A. K. Jónsson. Constraint-based attribute and interval planning. *Constraints*, 2003.

[373] J. Frank, et al. When gravity fails: Local search topology. *JAIR*, 1997.

[374] G. Fraser, et al. Plan execution in dynamic environments. In *Internat. Cognitive Robotics Workshop*. 2004.

[375] S. Fratini, et al. APSI-based deliberation in goal oriented autonomous controllers. In *ASTRA*, 2011.

[376] J. Fu, et al. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *IJCAI*, 2011.

[377] F. Fusier, et al. Video understanding for complex activity recognition. *Machine Vision and Applications*, 2007.

[378] Futur of Life Institute. Lethal Autonomous Weapons Pledge, 2018.

[379] Y. Gao, et al. Retrieval-augmented generation for large language models: A survey. *arXiv:2312.10997*, 2023.

[380] C. E. Garcia, et al. Model predictive control: theory and practice – a survey. *Automatica*, 1989.

[381] F. Garcia and P. Laborie. Hierarchisation of the search space in temporal planning. In *European Workshop on Planning*, 1995.

[382] R. García-Martínez and D. Borrajo. An integrated approach of learning, planning, and execution. *JIRS*, 2000.

[383] M. Garnelo, et al. Towards deep symbolic reinforcement learning. *arXiv:1609.05518*, 2016.

[384] C. R. Garrett, et al. Backward-forward search for manipulation planning. In *IROS*, 2015.

[385] C. R. Garrett, et al. Learning to rank for synthesizing planning heuristics. In *IJCAI*, 2016.

[386] C. R. Garrett, et al. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *arXiv:1608.01335*, 2016.

[387] C. R. Garrett, et al. Integrated task and motion planning. *ARCRAS*, 2021.

[388] A. Garrido. A temporal plannnig system for level 3 durative actions of PDDL2.1. In *AIPS Workshop on Planning for Temporal Domains*, 2002.

[389] A. Garrido and S. Jiménez. Learning temporal action models via constraint programming. In *ECAI*, 2020.

[390] T. Gedicke, et al. Flap for caos: Forward-looking active perception for clutter-aware object search. *IFAC Symposium on Intelligent Autonomous Vehicles*, 2016.

[391] H. Geffner. Functional Strips: A more flexible language for planning and problem solving. In *Logic-Based Artificial Intelligence*. Kluwer, 2000.

[392] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, 2013.

[393] T. Geffner and H. Geffner. Compact policies for fully observable non-deterministic planning as SAT. In *ICAPS*, 2018.

[394] C. Gehring, et al. Reinforcement learning for classical planning: Viewing heuristics as dense reward generators. In *ICAPS*, 2022.

[395] C. Geib and R. P. Goldman. A probabilistic plan recognition algorithm based on plan

tree grammars. *AIJ*, 2009.

[396] T. Geier and P. Bercher. On the decidability of HTN planning with task insertion. In *IJCAI*, 2011.

[397] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *JAIR*, 1996.

[398] A. Gerevini and L. Schubert. Discovering state constraints in DISCOPLAN: Some new results. In *AAAI*, Aug. 2000.

[399] A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs. In *AIPS*, 2002.

[400] A. Gerevini, et al. Planning through stochastic local search and temporal action graphs in LPG. *JAIR*, 2003.

[401] A. Gerevini, et al. Integrating planning and temporal reasoning for domains with durations and time windows. In *IJCAI*, 2005.

[402] A. Gerevini, et al. Combining domain-independent planning and HTN planning: The Duet planner. In *ECAI*, 2008.

[403] Z. Ghahramani. Learning dynamic bayesian networks. In *Summer School on Neural Networks*, 1997.

[404] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In *KR*, 1996.

[405] M. Ghallab. Responsible AI: requirements and challenges. *AI Perspectives*, 2019.

[406] M. Ghallab and H. Laruelle. Representation and control in IxTeT, a temporal planner. In *AIPS*, 1994.

[407] M. Ghallab and A. Mounir-Alaoui. Managing efficiently temporal relations through indexed spanning trees. In *IJCAI*, 1989.

[408] M. Ghallab, et al. Dealing with time in planning and execution monitoring. In *ISRR*, 1987.

[409] M. Ghallab, et al. PDDL–the planning domain definition language. Technical Report TR-98-003/TR-1165, Yale Center for Computational Vision and Control, 1998.

[410] M. Ghallab, et al. *Automated Planning: Theory and Practice*. Morgann Kaufmann, Oct. 2004.

[411] M. Ghallab, et al. *Automated Planning and Acting*. Cambridge University Press, 2016.

[412] M. Gharbi, et al. Combining symbolic and geometric planning to synthesize human-aware plans: toward more efficient combined search. In *IROS*, 2015.

[413] G. Ghiasi, et al. Scaling open-vocabulary image segmentation with image-level labels. In *ECCV*, 2022.

[414] A. Ghosh, et al. Exploring the frontier of vision-language models: A survey of current methodologies and future directions. *arXiv:2404.07214*, 2024.

[415] S. Ghosh, et al. ITS: An efficient limited-memory heuristic tree search algorithm. In *AAAI*, 1994.

[416] G. D. Giacomo and M. Favorito. Compositional approach to translate LTLf/LDLf into deterministic finite automata. In *ICAPS*, 2021.

[417] G. D. Giacomo and S. Rubin. Automata-theoretic foundations of FOND planning for LTLf and LDLf goals. In *IJCAI*, 2018.

[418] G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, 2013.

[419] G. D. Giacomo, et al. Timed trace alignment with metric temporal logic over finite traces. In *KR*, 2021.

[420] G. D. Giacomo, et al. LTLf synthesis as AND-OR graph search: Knowledge compilation at work. In L. D. Raedt, editor, *IJCAI*, 2022.

[421] Y. Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *ICML*, 1994.

[422] M. L. Gini, et al. Advances in autonomous robots for service and entertainment. *RAS*, 2010.

[423] E. Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *KR*, 2000.

[424] F. Giunchiglia. Using Abstrips abstractions – where do we stand? *AI Review*, 1999.

[425] F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, Sept. 1999.

[426] R. Givan, et al. Equivalence notions and model minimization in Markov decision processes. *AIJ*, 2003.

[427] A. Glaese, et al. Improving alignment of dialogue agents via targeted human judgements. *arXiv:2209.14375*, 2022.

[428] E. M. Gold. Complexity of automaton identification from given data. *Inf. Control.*, 1978.

[429] K. Golden, et al. Omnipotence without omniscience: Efficient sensor management for planning. In *AAAI*, 1994.

[430] R. Goldman. A semantics for HTN methods. In *ICAPS*, volume 19, 2009.

[431] R. Goldman, et al. Hard real-time mode logic synthesis for hybrid control: A CIRCA-based approach. In *AAAI Spring Symposium on Hybrid Systems and AI*, Mar. 1999. AAAI Tech. Report SS-99-05.

[432] R. Goldman, et al. A comparative analysis of plan repair in HTN planning. In *HPlan*, 2024.

[433] R. P. Goldman and U. Kuter. Hierarchical task network planning in Common Lisp: the case of SHOP3. In *European Lisp Symposium*, 2019.

[434] R. P. Goldman, et al. Dynamic abstraction planning. In *AAAI*, 1997.

[435] R. P. Goldman, et al. Using model checking to plan hard real-time controllers. In *AIPS Workshop on Model-Theoretic Approaches to Planning*, April 2000.

[436] R. P. Goldman, et al. Stable plan repair for state-space HTN planning. *HPlan*, 2020.

[437] M. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *JACM*, 1993.

[438] I. Goodfellow, et al. *Deep Learning*. MIT Press, 2016.

[439] I. Goodfellow, et al. Generative adversarial networks. *CACM*, 2020.

[440] M. Gopal. *Control Systems: Principles and Design*. McGraw-Hill, 1963.

[441] N. Goyal and D. Steiner. Graph neural networks for image classification and reinforcement learning using graph representations. *arXiv:2203.03457*, 2022.

[442] P. Goyal, et al. Using natural language for reward shaping in reinforcement learning. *arXiv:1903.02020*, 2019.

[443] A. Gragera, et al. A planning approach to repair domains with incomplete action effects. In *ICAPS*, 2023.

[444] M. Grand, et al. Tempamlsi: Temporal action model learning based on STRIPS translation. In *ICAPS*, volume 32, 2022.

[445] I. Greenberg, et al. Train hard, fight easy: Robust meta reinforcement learning. In *NeurIPS*, 2023.

[446] P. Gregory, et al. A meta-CSP model for optimal planning. In *Abstraction, Reformulation, and Approximation*. Springer, 2007.

[447] P. Gregory, et al. Planning modulo theories: Extending the planning paradigm. In *ICAPS*, 2012.

[448] I. Grondman, et al. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *SMC*, 2012.

[449] S. Gu, et al. Continuous deep Q-learning with model-based acceleration. In *ICML*, volume 48, 2016.

[450] S. Gu, et al. Teams-rl: Teaching llms to teach themselves better instructions via reinforcement learning. *arXiv:2403.08694*, 2024.

[451] L. Guan, et al. Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. *arXiv:2305.14909*, 2023.

[452] C. Guestrin, et al. Efficient solution algorithms for factored MDPs. *JAIR*, 2003.

[453] C. Guestrin, et al. Solving factored MDPs with continuous and discrete variables. In *UAI*, 2004.

[454] A. Guez and J. Pineau. Multi-tasking SLAM. In *ICRA*, 2010.

[455] E. Guizzo. Kiva Systems. *IEEE Spectrum*, July 2008.

[456] A. Gupta, et al. Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. *arXiv:1910.11956*, 2019.

[457] S. Gutstein and E. Stump. Reduction of catastrophic forgetting with transfer learning and ternary output codes. In *IJCNN*, 2015.

[458] A. Hafiz. A survey of deep Q-networks used for reinforcement learning: State of the art. *Intelligent Communication Technologies and Virtual Mobile Networks*, 2023.

[459] M. Hägele, et al. Industrial robotics. In *Handbook of Robotics*. Springer, 2008.

[460] M. Hahn. Theoretical limitations of self-attention in neural sequence models. *arXiv:1906.06755*, 2020.

[461] D. Hähnel, et al. GOLEX—bridging the gap between logic (GOLOG) and a real robot. In *KI*. 1998.

[462] W. L. Hamilton. *Graph Representation Learning*. Morgan & Claypool publishers, 2020.

[463] K. J. Hammond. Explaining and repairing plans that fail. *AIJ*, 1990.

[464] S. Hanks and R. J. Firby. Issues and architectures for planning and execution. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[465] S. Hanks and D. S. Weld. A domain-independent algorithm for plan adaptation. *JAIR*, 1995.

[466] E. A. Hansen. Indefinite-horizon pomdps with action-based termination. In *AAAI*, 2007.

[467] E. A. Hansen. Suboptimality bounds for stochastic shortest path problems. In *UAI*, 2011.

[468] E. A. Hansen and R. Zhou. Anytime heuristic search. *JAIR*, 2007.

[469] E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *AIJ*, 2001.

[470] P. Hart and A. Knoll. Graph neural networks and reinforcement learning for behavior generation in semantic environments. In *IEEE Intelligent Vehicles Symposium*, 2020.

[471] P. E. Hart, et al. A formal basis for the heuristic determination of minimum cost paths. *SMC*, 1968.

[472] P. E. Hart, et al. Correction to a formal basis for the heuristic determination of minimum cost paths. *SIGART Bulletin*, 1972.

[473] A. Harutyunyan, et al. Hindsight credit assignment. *NeurIPS*, 2019.

[474] M. Hasanbeig, et al. Deepsynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. In *AAAI*, 2021.

[475] P. Haslum. Admissible makespan estimates for PDDL2.1 temporal planning. In *HDIP*, 2009.

[476] P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *AIPS*, 2000.

[477] P. Haslum and H. Geffner. Heuristic plannnig with time and resources. In *ECP*, 2001.

[478] P. Haslum, et al. New admissible heuristics for domain-independent planning. In *AAAI*, 2005.

[479] P. Haslum, et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*, volume 7, 2007.

[480] P. Haslum, et al. Extending classical planning with state constraints: Heuristics and search for optimal planning. *JAIR*, June 2018.

[481] P. Haslum, et al. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on AI and ML. Morgan & Claypool Publishers, 2019.

[482] K. Hauser. Task planning with continuous actions and nondeterministic motion planning queries. In *AAAI Workshop on Bridging the Gap between Task and Motion Planning*, 2010.

[483] K. Hauser and J.-C. Latombe. Integrating task and PRM motion planning: Dealing with many infeasible motion planning queries. In *ICAPS*, 2009.

[484] M. Hauskrecht and B. Kveton. Linear program approximations for factored continuous-state markov decision processes. *NeurIPS*, 2003.

[485] M. Hauskrecht, et al. Hierarchical solution of Markov decision processes using macro-actions. In *UAI*, 1998.

[486] N. Hawes. A survey of motivation frameworks for intelligent systems. *AIJ*, 2011.

[487] J.-B. Hayet, et al. Motion planning for maintaining landmarks visibility with a differential drive robot. *RAS*, 2014.

[488] F. Heintz, et al. Bridging the sense-reasoning gap: DyKnow – stream-based middleware for knowledge processing. *Advanced Engineering Informatics*, 2010.

[489] M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, 2002.

[490] M. Helmert. The Fast Downward planning system. *JAIR*, 2006.

[491] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *AIJ*, 2009.

[492] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 2009.

[493] M. Helmert and H. Geffner. Unifying the causal graph and additive heuristics. In *ICAPS*, 2008.

[494] M. Helmert, et al. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 2007.

[495] M. Helmert, et al. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, 2008.

[496] M. Helmert, et al. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM*, 2014.

[497] M. Helmert, et al. On the complexity of heuristic synthesis for satisficing classical planning: Potential heuristics and beyond. In *ICAPS*, 2022.

[498] P. Hérail. *Learning Hierarchical Models from Demonstrations for Deliberate Planning and Acting*. PhD thesis, University of Toulouse, 2024.

[499] T. Hester and P. Stone. TEXPLORE: real-

time sample-efficient reinforcement learning for robots. *ML*, 2013.

[500] T. Hester, et al. Deep Q-learning from demonstrations. In *AAAI*, 2018.

[501] P. Hitzler. A review of the semantic web field. *Communications of the ACM*, 2021.

[502] P. Hitzler and M. Wendt. A uniform approach to logic programming semantics. *Theory and Practice of Logic Programming*, 2005.

[503] J. Ho and S. Ermon. Generative adversarial imitation learning. *NeurIPS*, 2016.

[504] H. Hoang, et al. Hierarchical plan representations for encoding strategic game AI. In *AIIDE*, 2005.

[505] J. Hoey, et al. SPUDD: Stochastic planning using decision diagrams. In *UAI*, 1999.

[506] J. Hoffmann. The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *JAIR*, 2003.

[507] J. Hoffmann. Where "ignoring delete lists" works: local search topology in planning benchmarks. *JAIR*, 2005.

[508] J. Hoffmann and R. Brafman. Contingent planning via heuristic forward search with implicit belief states. In *ICAPS*, 2005.

[509] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 2001.

[510] J. Hoffmann, et al. Ordered landmarks in planning. *JAIR*, 2004.

[511] C. Hogg, et al. Learning hierarchical task models from input traces. *CI*, 2016.

[512] D. Höller. Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages. In *ICAPS*, volume 31, 2021.

[513] D. Höller, et al. Assessing the expressivity of planning formalisms through the comparison to formal languages. In *ICAPS*, 2016.

[514] D. Höller, et al. A generic method to guide HTN progression search with classical heuristics. In *ICAPS*, 2018.

[515] D. Höller, et al. HDDL: An extension to PDDL for expressing hierarchical planning problems. In *AAAI*, 2020.

[516] D. Höller, et al. HTN plan repair via model transformation. In *KI*, 2020.

[517] D. Höller, et al. Compiling HTN plan verification problems into HTN planning problems. In *ICAPS*, 2022.

[518] S. Hongeng, et al. Video-based event recognition: activity representation and probabilistic recognition methods. *Computer Vision and Image Understanding*, 2004.

[519] J. N. Hooker. Operations research methods in constraint programming. In F. Rossi, et al., editors, *Handbook of Constraint Programming*. Elsevier, 2006.

[520] B. Horling, et al. Distributed sensor network for real time tracking. In *AAMAS*, 2001.

[521] S. S. E. Horowitz and S. Rajasakaran. *Computer Algorithms*. W.H. Freeman, 1996.

[522] R. A. Howard. *Dynamic Probabilistic Systems*. Wiley, 1971.

[523] H. Hu and D. Sadigh. Language instructed reinforcement learning for human-ai coordination. In *ICML*. PMLR, 2023.

[524] W. Hu, et al. Bidirectional projection network for cross dimension scene understanding. In *CVPR*, 2021.

[525] Y. Hu, et al. What can knowledge bring to machine learning?—a survey of low-shot learning for structured data. *ACM Transactions on Intelligent Systems and Technology*, 2022.

[526] B. Huang, et al. AdaRL: What, where, and how to adapt in transfer reinforcement learning. *ICLR*, 2022.

[527] J. Huang and K. C.-C. Chang. Towards reasoning in large language models: A survey. *arXiv:2212.10403*, 2022.

[528] R. Huang, et al. An optimal temporally expressive planner: Initial results and application to P2P network optimization. In *ICAPS*, 2009.

[529] R. Huang, et al. SAS+ planning as satisfiability. *JAIR*, 2012.

[530] W. Huang, et al. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *ICML*, 2022.

[531] I. Hwang, et al. A survey of fault detection, isolation, and reconfiguration methods. *TCST*, 2010.

[532] T. Ibaraki. Theoretical comparision of search strategies in branch and bound. *International Journal of Computer and Information Sciences*, 1976.

[533] B. Ichter, et al. Learning sampling distributions for robot motion planning. In *ICRA*, 2018.

[534] O. Ilghami and D. S. Nau. A general approach to synthesize problem-

specific planners. Technical Report CS-TR-4597, UMIACS-TR-2004-40, University of Maryland, Oct. 2003.

[535] G. E. Imaz and M. Ghallab. A practically efficient and almost linear unification algorithm. *AIJ*, 1988.

[536] M. D. Ingham, et al. A reactive model-based programming language for robotic space explorers. In *i-SAIRAS*, 2001.

[537] F. Ingrand. ProSkill: A formal skill language for acting in robotics. *arXiv:2403.07770*, 2024.

[538] F. Ingrand and O. Despouys. Extending procedural reasoning toward robot actions planning. In *ICRA*, 2001.

[539] F. Ingrand and M. Ghallab. Deliberation for Autonomous Robots: A Survey. *AIJ*, 2017.

[540] F. Ingrand, et al. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, 1996.

[541] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

[542] M. Iovino, et al. A survey of behavior trees in robotics and AI. *RAS*, 2022.

[543] M. Iovino, et al. A framework for learning behavior trees in collaborative robotic applications. In *CASE*, 2023.

[544] D. Isla. Handling complexity in the Halo 2 AI. In *GDC*, 2005.

[545] M. Iwen and A. D. Mali. Distributed graphplan. In *ICTAI*, 2002.

[546] M. Jahangirian, et al. Simulation in manufacturing and business: A review. *European Jour. of Operational Research*, 2010.

[547] L. Jaillet and T. Siméon. A PRM-based motion planner for dynamically changing environments. In *IROS*, 2004.

[548] K. Jensen. *Coloured Petri Nets*. Springer, 1992.

[549] K. Jensen, et al. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *Internat. Jour. on Software Tools for Technology Transfer*, 2007.

[550] R. Jensen and M. Veloso. OBDD-based universal planning for synchronized agents in non-deterministic domains. *JAIR*, 2000.

[551] R. Jensen, et al. Guided symbolic universal planning. In *ICAPS*, June 2003.

[552] R. M. Jensen, et al. OBDD-based optimistic and strong cyclic adversarial planning. In *ECP*, 2001.

[553] S. Ji, et al. 3d convolutional neural networks for human action recognition. *PAMI*, 2012.

[554] Y. Jiang, et al. Language as an abstraction for hierarchical deep reinforcement learning. In *NeurIPS*, 2019.

[555] Z. Jiang, et al. Active retrieval augmented generation. *arXiv:2305.06983*, 2023.

[556] S. Jiménez, et al. A review of machine learning for automated planning. *KER*, 2012.

[557] S. Jo and I. Trummer. Smart: Automatically scaling down language models with accuracy guarantees for reduced processing fees. *arXiv:2403.13835*, 2024.

[558] A. K. Jónsson, et al. Planning in interplanetary space: Theory and practice. In *AIPS*, 2000.

[559] P. Jonsson, et al. Computational complexity of relating time points and intervals. *AIJ*, 1999.

[560] M. Jordan and A. Perez. Optimal bidirectional rapidly-exploring random trees. Technical report, MIT-CSAIL-TR-021, 2013.

[561] M. Jovanović and P. Voss. Towards incremental learning in large language models: A critical review, 2024. Online report.

[562] B. Juba and R. Stern. Learning probably approximately complete and safe action models for stochastic worlds. In *AAAI*, 2022.

[563] B. Juba, et al. Safe learning of lifted action models. In *KR*, 2021.

[564] F. Kabanza, et al. Planning control rules for reactive agents. *AIJ*, 1997.

[565] L. P. Kaelbling. Learning to achieve goals. In *IJCAI*, 1993.

[566] L. P. Kaelbling and T. Lozano-Perez. Hierarchical task and motion planning in the now. In *ICRA*, 2011.

[567] L. P. Kaelbling and T. Lozano-Perez. Integrated task and motion planning in belief space. *IJRR*, 2013.

[568] L. P. Kaelbling and T. Lozano-Perez. Implicit belief-space pre-images for hierarchical planning and execution. In *ICRA*, 2016.

[569] L. P. Kaelbling, et al. Reinforcement learning: A survey. *JAIR*, 1996.

[570] L. P. Kaelbling, et al. Planning and acting in partially observable stochastic domains. *AIJ*, 1998.

[571] S. Kakade and J. Langford. Approximately optimal approximate reinforcement learn-

ing. In *ICML*, 2002.

[572] S. Kambhampati. On the utility of systematicity: Understanding the trade-offs between redundancy and commitment in partial-order planning. In *IJCAI*, 1993.

[573] S. Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AIMag*, 1997.

[574] S. Kambhampati. On the relations between intelligent backtracking and failure-driven explanation-based learning in constraint satisfaction and planning. *AIJ*, 1998.

[575] S. Kambhampati. Are we comparing Dana and Fahiem or SHOP and TLPlan? A critique of the knowledge-based planning track at ICP, 2003.

[576] S. Kambhampati. Polanyi's revenge and AI's new romance with tacit knowledge. *CACM*, 2021.

[577] S. Kambhampati and J. A. Hendler. A validation-structure-based theory of plan modification and reuse. *AIJ*, 1992.

[578] S. Kambhampati and B. Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In *ECP*, 1995.

[579] S. Kambhampati and S. W. Yoon. Explanation-based learning for planning. In C. Sammut and G. I. Webb, editors, *Encyclopedia of Machine Learning*. Springer, 2010.

[580] S. Kambhampati, et al. Failure driven dynamic search control for partial order planners: An explanation based approach. *AIJ*, 1996.

[581] S. Kambhampati, et al. Hybrid planning for partially hierarchical domains. In *AAAI*, 1998.

[582] O. Kanoun, et al. Planning foot placements for a humanoid robot: A problem of inverse kinematics. *IJRR*, 2011.

[583] E. Karabaev and O. Skvortsova. A heuristic search algorithm for solving first-order MDPs. In *UAI*, 2005.

[584] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *IJRR*, 2011.

[585] R. Karia and S. Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *AAAI*, 2021.

[586] L. Karlsson, et al. To secure an anchor – A recovery planning approach to ambiguity in perceptual anchoring. *AI Communincations*, 2008.

[587] E. Karpas, et al. Temporal landmarks: What must happen, and when. In *ICAPS*, 2015.

[588] M. Katz and C. Domshlak. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, 2008.

[589] M. Katz and C. Domshlak. Structural-pattern databases. In *ICAPS*, 2009.

[590] H. Kautz and J. Allen. Generalized plan recognition. In *AAAI*, 1986.

[591] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *AAAI*, 1996.

[592] H. A. Kautz, et al., editors. *Synthesis and Planning*, Dagstuhl Seminar Proceedings, 2006.

[593] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration for fast path planning. In *ICRA*, 1994.

[594] H. Kazerooni. Exoskeletons for human performance augmentation. In *Handbook of Robotics*. Springer, 2008.

[595] M. Kearns, et al. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *ML*, 2002.

[596] G. Kelleher and A. G. Cohn. Automatically synthesising domain constraints from operator descriptions. In *ECAI*, 1992.

[597] T. Keller and P. Eyerich. PROST: Probabilistic planning based on UCT. In *ICAPS*, 2012.

[598] T. Keller and M. Helmert. Trial-based heuristic tree search for finite horizon MDPs. In *ICAPS*, volume 23, 2013.

[599] H. Kerzner. *Project management: a systems approach to planning, scheduling, and controlling*. John Wiley & Sons, 2017.

[600] M. A.-M. Khan, et al. A systematic review on reinforcement learning-based robotics within the last decade. *IEEE Access*, 2020.

[601] L. Khatib, et al. Temporal constraint reasoning with preferences. In *IJCAI*, 2001.

[602] O. Khatib. The potential field approach and operational space formulation in robot control. In *Adaptive and Learning Systems: Theory and Applications*. Springer, 1986.

[603] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *IJRR*, 1986.

[604] S. Kiesel and W. Ruml. Planning under temporal uncertainty using hindsight optimization. In *ICAPS Wksp. on Planning and Robotics*, 2014.

[605] B. Kim and J. Pineau. Socially adaptive path planning in human environments us-

ing inverse reinforcement learning. *Internat. Jour. of Social Robotics*, 2016.
[606] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.
[607] Z. K. Kingston, et al. Sampling-based methods for motion planning with constraints. *ARCRAS*, May 2018.
[608] B. R. Kiran, et al. Deep reinforcement learning for autonomous driving: A survey. *T-ITS*, 2021.
[609] P. Kissmann and S. Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *KI*. 2009.
[610] G. Klein and M. Mouhoub. Solving temporal constraints using neural networks. In *Internat. Conf. on Artificial Intelligence (IC-AI)*, 2002.
[611] T. Klößner and J. Hoffmann. Pattern databases for stochastic shortest path problems. In *SOCS*, 2021.
[612] T. Klößner, et al. Pattern databases for goal-probability maximization in probabilistic planning. In *ICAPS*, 2021.
[613] T. Klößner, et al. Cost partitioning heuristics for stochastic shortest path problems. In *ICAPS*, 2022.
[614] T. Klößner, et al. Cartesian abstractions and saturated cost partitioning in probabilistic planning. In *ECAI*, 2023.
[615] T. Klößner, et al. A theory of merge-and-shrink for stochastic shortest path problems. In *ICAPS*, 2023.
[616] R. Knight, et al. Casper: space exploration through continuous planning. *IEEE Intelligent Systems*, 2001.
[617] C. A. Knoblock. Automatically generating abstractions for planning. *AIJ*, 1994.
[618] C. A. Knoblock and Q. Yang. Relating the performance of partial-order planning algorithms to domain features. *SIGART Bulletin*, 1995.
[619] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *AIJ*, 1975.
[620] J. Kober and J. Peters. Policy search for motor primitives in robotics. *ML*, 2011.
[621] J. Kober, et al. Reinforcement learning to adjust robot movements to new situations. In *RSS*, 2010.
[622] J. Kober, et al. Reinforcement learning in robotics: A survey. *IJRR*, 2013.
[623] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, 2006.
[624] J. Koehler. Planning under resource constraints. In *ECAI*, 1998.
[625] J. Koehler. Handling of conditional effects and negative goals in IPP. Technical Report 128, Albert-Ludwigs-Universität Freiburg, 1999.
[626] S. Koenig. Minimax real-time heuristic search. *AIJ*, 2001.
[627] S. Koenig and R. Simmons. Solving robot navigation problems with initial pose uncertainty using real-time heuristic search. In *AIPS*, 1998.
[628] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
[629] A. Kolobov and D. Weld. ReTrASE: integrating paradigms for approximate probabilistic planning. In *IJCAI*, 2009.
[630] A. Kolobov, et al. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *AAAI*, Apr. 2010.
[631] A. Kolobov, et al. Heuristic search for generalized stochastic shortest path MDPs. In *ICAPS*, 2011.
[632] A. Kolobov, et al. Reverse iterative deepening for finite-horizon mdps with large branching factors. In *ICAPS*, volume 22, 2012.
[633] A. Kolobov, et al. Stochastic shortest path MDPs with dead ends. *HSDIP*, 2012.
[634] V. R. Konda and V. S. Borkar. Actor-critic–type learning algorithms for Markov decision processes. *SIAM Jour. on Control and Optimization*, 1999.
[635] Y. Kong and Y. Fu. Human action recognition and prediction: A survey. *IJCV*, 2022.
[636] G. Konidaris, et al. Robot learning from demonstration by constructing skill trees. *IJRR*, 2012.
[637] G. Konidaris, et al. From skills to symbols: Learning symbolic representations for abstract high-level planning. *JAIR*, 2018.
[638] K. Konolige, et al. Navigation in hybrid metric-topological maps. In *ICRA*, 2011.
[639] R. Korf. Real-time heuristic search. *AIJ*, 1990.
[640] R. E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *AIJ*, 1985.
[641] R. E. Korf. Planning as search: A quantitative approach. *AIJ*, 1987.
[642] R. E. Korf. Linear-space best-first search. *AIJ*, 1993.
[643] P. Kormushev, et al. Reinforcement learning in robotics: Applications and real-

world challenges. *Robotics*, 2013.

[644] M. Koubarakis. From local to global consistency in temporal constraint networks. *TCS*, 1997.

[645] A. Krizhevsky, et al. Imagenet classification with deep convolutional neural networks. *CACM*, 2017.

[646] O. Kroemer, et al. A review of robot learning for manipulation: Challenges, representations, and algorithms. *JMLR*, 2021.

[647] V. Krüger, et al. The meaning of action: a review on action recognition and mapping. *Advanced Robotics*, 2007.

[648] J. J. Kuffner and S. M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *ICRA*, 2000.

[649] S. Kuindersma, et al. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 2016.

[650] B. Kuipers and Y.-T. Byun. A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations. *RAS*, 1991.

[651] B. Kuipers, et al. Local metrical and global topological maps in the hybrid spatial semantic hierarchy. In *ICRA*, 2004.

[652] T. D. Kulkarni, et al. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *NeurIPS*, 2016.

[653] S. Kumar. Balancing a cartpole system with reinforcement learning–a tutorial. *arXiv:2006.04938*, 2020.

[654] V. Kumar and L. Kanal. A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *AIJ*, Mar. 1983.

[655] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *IEEE Symposium on Logic in Computer Science*, 2001.

[656] O. Kupferman, et al. Open systems in reactive environments: Control and synthesis. In *CONCUR*, 2000.

[657] H. Kurutach, et al. Learning plannable representations with causal infogan. In *NeurIPS*, 2018.

[658] U. Kuter and D. Nau. Using domain-configurable search control for probabilistic planning. In *AAAI*, 2005.

[659] U. Kuter, et al. Using classical planners to solve nondeterministic planning problems. In *ICAPS*, Sept. 2008.

[660] U. Kuter, et al. Task decomposition on abstract states, for planning under nondeterminism. *AIJ*, 2009.

[661] J. Kvarnström and P. Doherty. TALplanner: A temporal logic based forward chaining planner. *AMAI*, 2001.

[662] B. Kveton, et al. Solving factored MDPs with hybrid state and action variables. *JAIR*, 2006.

[663] M. Kwon, et al. Reward design with language models. *arXiv:2303.00001*, 2023.

[664] P. Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *AIJ*, 2003.

[665] P. Laborie and M. Ghallab. Planning with sharable resource constraints. In *IJCAI*, 1995.

[666] M. G. Lagoudakis and R. Parr. Model-free least-squares policy iteration. *NeurIPS*, 2001.

[667] M. G. Lagoudakis and R. Parr. Reinforcement learning as classification: Leveraging modern classifiers. In *ICML*, 2003.

[668] F. Lagriffoul and B. Andres. Combining task and motion planning: A culprit detection problem. *IJRR*, 2016.

[669] F. Lagriffoul, et al. Combining task and motion planning is not always a good idea. In *RSS Workshop on Combined Robot Motion Planning and AI Planning for Practical Applications*, 2013.

[670] F. Lagriffoul, et al. Efficiently combining task and motion planning using geometric constraints. *IJRR*, 2014.

[671] J. Laird, et al. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies*, volume 11. Springer Science & Business Media, 2012.

[672] L. Lamanna and L. Serafini. Action model learning from noisy traces: a probabilistic approach. In *ICAPS*, 2024.

[673] L. Lamanna, et al. On-line learning of planning domains from sensor data in pal: Scaling up to large state spaces. In *AAAI*, 2021.

[674] L. Lamanna, et al. Online learning of action models for PDDL planning. In *IJCAI*, 2021.

[675] L. Lamanna, et al. Online grounding of symbolic planning domains in unknown environments. In *KR*, 2022.

[676] L. Lamanna, et al. Learning to act for perceiving in partially unknown environments. In *IJCAI*, 2023.

[677] L. Lamanna, et al. Planning for learning object properties. In *AAAI*, 2023.

[678] M. Lan, et al. A modular mission management system for micro aerial vehicles. In *IEEE 14th Internat. Conf. on Control and Automation*, 2018.

[679] S. Lange, et al. Batch reinforcement learning. In *Reinforcement learning: State-of-the-art*. Springer, 2012.

[680] P. Langley. Learning hierarchical problem networks for knowledge-based planning. In *Internat. Conf. on Inductive Logic Programming*, 2022.

[681] P. Langley and D. Choi. Learning recursive control programs from problem solving. *JMLR*, 2006.

[682] P. Langley, et al. Hierarchical problem networks for knowledge-based planning. In *Annual Conf. on Advances in Cognitive Systems*, 2021.

[683] C. Laporte and T. Arbel. Efficient discriminant viewpoint selection for active Bayesian recognition. *IJRR*, 2006.

[684] J.-C. Latombe. *Robot Motion Planning*. Kluwer, Boston, MA, 1991.

[685] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[686] S. M. LaValle and J. J. Kuffner Jr. Randomized kinodynamic planning. *IJRR*, 2001.

[687] A. Lazaric. Transfer in reinforcement learning: a framework and a survey. In *Reinforcement Learning: State-of-the-Art*. Springer, 2012.

[688] A. Lazaridis, et al. Deep reinforcement learning: A state-of-the-art walkthrough. *JAIR*, 2020.

[689] X. Le Guillou, et al. Chronicles for on-line diagnosis of distributed systems. In *ECAI*, volume 8, 2008.

[690] H. Lee, et al. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv:2309.00267*, 2023.

[691] J. Lee, et al. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 2020.

[692] J. B. Lee, et al. Temporal network representation learning. *arXiv:1904.06449*, 2019.

[693] M. Lee and C. W. Anderson. Can a reinforcement learning agent practice before it starts learning? In *IJCNN*, 2017.

[694] S. Lemai-Chenevier and F. Ingrand. Interleaving temporal planning and execution in robotics domains. In *AAAI*, 2004.

[695] B. León, et al. Opengrasp: a toolkit for robot grasping simulation. In *Internat. Conf. on Simulation, Modeling, and Programming for Autonomous Robots*, 2010.

[696] C. Lesire and F. Pommereau. ASPiC: an acting system based on skill Petri net composition. In *IROS*, 2018.

[697] V. Lesser, et al. Evolution of the gpgp/tæms domain-independent coordination framework. *JAAMAS*, 2004.

[698] H. Levesque, et al. GOLOG: A logic programming language for dynamic domains. *Jour. of Logic Programming*, 1997.

[699] S. Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv:1805.00909*, 2018.

[700] S. Levine and V. Koltun. Guided policy search: deep RL with importance sampled policy gradient. In *ICML*, 2013.

[701] S. Levine, et al. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv:2005.01643*, 2020.

[702] S. J. Levine and B. C. Williams. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*, Nov. 2014.

[703] B. Li, et al. Language-driven semantic segmentation. *arXiv:2201.03546*, 2022.

[704] C. Li, et al. Multimodal foundation models: From specialists to general-purpose assistants. *arXiv:2309.10020*, 2023.

[705] H. X. Li and B. C. Williams. Generative planning for hybrid systems based on flow tubes. In *ICAPS*, 2008.

[706] J. Li, et al. Scalable rail planning and replanning: Winning the 2020 Flatland Challenge. In *ICAPS*, volume 31, 2021.

[707] K. Li, et al. Emergent world representations: Exploring a sequence model trained on a synthetic task. *arXiv:2210.13382*, 2022.

[708] L. Li and M. L. Littman. Lazy approximation for solving continuous finite-horizon mdps. In *AAAI*, 2005.

[709] M. Li, et al. API-bank: A comprehensive benchmark for tool-augmented llms. *arXiv:2304.08244*, 2023.

[710] R. Li. *Automating Hierarchical Task Network Learning*. PhD thesis, University of Maryland, June 2024.

[711] Z. Li, et al. Reinforcement learning for robust parameterized locomotion control of bipedal robots. In *ICRA*, 2021.

[712] J. Liang, et al. Code as policies: Language model programs for embodied control. In *ICRA*, 2023.

[713] V. Liatsos and B. Richard. Scalability in

planning. In *ECP*, 1999.

[714] A. O. Liberman, et al. Learning first-order symbolic planning representations that are grounded. *arXiv:2204.11902*, 2022.

[715] V. Lifschitz. On the semantics of STRIPS. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans*. Morgan Kaufmann, 1987.

[716] G. Ligozat. On generalized interval calculi. In *AAAI*, 1991.

[717] M. Likhachev, et al. Planning for Markov decision processes with sparse stochasticity. In *NeurIPS*, volume 17, 2004.

[718] T. P. Lillicrap, et al. Continuous control with deep reinforcement learning. *arXiv:1509.02971*, 2016.

[719] M. H. Lim, et al. Sparse tree search optimality guarantees in pomdps with continuous observation spaces. *arXiv:1910.04332*, 2019.

[720] K. Lin, et al. Text2Motion: From natural language instructions to feasible plans. *arXiv:2303.12153*, 2023.

[721] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Jour.*, 1965.

[722] N. Lipovetzky and H. Geffner. Best-first width search: Exploration and exploitation in classical planning. In *AAAI*, 2017.

[723] I. Little and S. Thiébaux. Probabilistic planning vs. replanning. In *ICAPS Wksp. on the International Planning Competition*, 2007.

[724] I. Little, et al. Prottle: A probabilistic temporal planner. In *AAAI*, 2005.

[725] M. L. Littman, et al. Gathering Strength, Gathering Storms: The One Hundred Year Study on Artificial Intelligence (AI100). Technical report, Stanford University, 2021.

[726] B. Liu, et al. LLM+P: Empowering large language models with optimal planning proficiency. *arXiv:2304.11477*, 2023.

[727] H. Liu, et al. Darts: Differentiable architecture search. *arXiv:1806.09055*, 2018.

[728] Y. Liu and S. Koenig. Functional value iteration for decision-theoretic planning with general utility functions. In *AAAI*, 2006.

[729] Y. Liu and S. Koenig. Functional value iteration for decision-theoretic planning with general utility functions. In *AAAI*, 2006.

[730] Y. Liu, et al. Learning search-space specific heuristics using neural networks. *arXiv*, 2023.

[731] I. Lluvia, et al. Active mapping and robot exploration: A survey. *Sensors*, 2021.

[732] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *JAIR*, 1999.

[733] D. Long and M. Fox. Exploiting a graphplan framework in temporal planning. In *ICAPS*, 2003.

[734] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *JAIR*, 2003.

[735] D. Long, et al. The AIPS-98 planning competition. *AIMag*, 2000.

[736] A. Lotem and D. S. Nau. New advances in GraphHTN: Identifying independent subproblems in large HTN domains. In *AIPS*, Apr. 2000.

[737] A. Lotem, et al. Using planning graphs for solving HTN problems. In *AAAI*, 1999.

[738] D. Lotinac and A. Jonsson. Constructing hierarchical task models using invariance analysis. In *ECAI*. IOS Press, 2016.

[739] T. Lozano-Pérez and L. P. Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In *IROS*, 2013.

[740] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *CACM*, Oct. 1979.

[741] J. Luketina, et al. A survey of reinforcement learning informed by natural language. *arXiv:1906.03926*, 2019.

[742] L. Ly and Y.-H. R. Tsai. Autonomous exploration, reconstruction, and surveillance of 3D environments aided by deep learning. In *ICRA*, 2019.

[743] K. M. Lynch and F. C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

[744] Y. J. Ma, et al. Eureka: Human-level reward design via coding large language models. *arXiv:2310.12931*, 2023.

[745] J. MacGlashan, et al. Interactive learning from policy-dependent human feedback. In *ML*, 2017.

[746] M. G. Madden and T. Howley. Transfer of experience between reinforcement learning environments with progressive difficulty. *Artificial Intelligence Review*, 2004.

[747] M. C. Magnaguagno, et al. HyperTensioN and total-order forward decomposition optimizations. *arXiv:2207.00345*, 2022.

[748] S. Mahadevan and J. Connell. Automatic

programming of behavior-based robots using reinforcement learning. *AIJ*, 1992.
[749] S. Maliah, et al. Partially observable online contingent planning using landmark heuristics. In *ICAPS*, 2014.
[750] J. Malik and T. Binford. Reasoning in time and space. In *IJCAI*, 1983.
[751] P. Mallick, et al. Reinforcement learning using expectation maximization based guided policy search for stochastic dynamics. *Neurocomputing*, 2022.
[752] M. Mansouri and F. Pecora. A robot sets a table: a case for hybrid reasoning with different types of knowledge. *JETAI*, 2016.
[753] J. Marecki, et al. A fast analytical algorithm for mdps with continuous state spaces. In *AAMAS Workshop on Game Theoretic and Decision Theoretic Agents*, 2006.
[754] B. Marthi, et al. Angelic semantics for high-level actions. In *ICAPS*, 2007.
[755] B. Marthi, et al. Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS*, 2008.
[756] B. M. Marthi, et al. Concurrent hierarchical reinforcement learning. In *AAAI*, 2005.
[757] A. Marzinotto, et al. Towards a unified behavior trees framework for robot control. In *ICRA*, 2014.
[758] N. Maslej, et al. The AI Index Annual Report. Technical report, Institute for Human-Centered AI, Stanford University, 2024.
[759] M. T. Mason. *Mechanics of robotic manipulation*. MIT press, 2001.
[760] M. T. Mason. Toward robotic manipulation. *ARCRAS*, 2018.
[761] M. T. Mason and J. K. Salisbury Jr. *Robot hands and the mechanics of manipulation*. The MIT Press, Cambridge, MA, 1985.
[762] M. Mateas and A. Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 2002.
[763] R. Mattmüller, et al. Pattern database heuristics for fully observable nondeterministic planning. In *ICAPS*, 2010.
[764] Mausam and A. Kolobov. *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool, 2012.
[765] Mausam and D. Weld. Concurrent probabilistic temporal planning. In *ICAPS*, 2005.
[766] Mausam and D. Weld. Probabilistic temporal planning with uncertain durations. In *AAAI*, 2006.
[767] Mausam and D. Weld. Planning with durative actions in stochastic domains. *JAIR*, 2008.
[768] Mausam, et al. A hybridized planner for stochastic domains. In *IJCAI*, 2007.
[769] S. McAleer, et al. Solving the Rubik's cube without human knowledge. *arXiv:1805.07470*, 2018.
[770] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *AAAI*, July 1991.
[771] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh Univ. Press, 1969.
[772] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 1943.
[773] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 1982.
[774] D. McDermott. A reactive plan language. Technical Report YALEU/CSD/RR 864, Yale Univ., 1991.
[775] J. McDonald, et al. Great power, great responsibility: Recommendations for reducing energy for training language models. *arXiv:2205.09646*, 2022.
[776] C. McGann, et al. A deliberative architecture for AUV control. In *ICRA*, 2008.
[777] S. A. McIlraith and T. C. Son. Adapting GOLOG for composition of semantic web services. In *KR*, 2002.
[778] H. B. McMahan and G. J. Gordon. Fast exact planning in Markov decision processes. In *ICAPS*, 2005.
[779] J. McMahon and E. Plaku. Robot motion planning with task specifications via regular languages. *Robotica*, 2017.
[780] J. Med, et al. Weak and strong reversibility of non-deterministic actions: Universality and uniformity. In *ICAPS*, 2024.
[781] N. Mehta, et al. Automatic induction of maxq hierarchies. In *NIPS Workshop: Hierarchical Organization of Behavior*, 2007.
[782] I. Meiri. Faster constraint satisfaction algorithms for temporal reasoning. Tech. report R-151, UC Los Angeles, 1990.
[783] M. R. Mendonça, et al. Graph-based skill acquisition for reinforcement learning. *CSUR*, 2019.

[784] A. Menif, et al. SHPE: HTN planning for video games. In *Workshop on Computer Games*, 2014.

[785] W. Merrill, et al. Provable limitations of acquiring meaning from ungrounded form: What will future language models understand? *Transactions of the Association for Computational Linguistics*, 2021.

[786] N. Meuleau and R. I. Brafman. Hierarchical heuristic forward search in stochastic domains. In *IJCAI*, 2007.

[787] N. Meuleau, et al. A heuristic search approach to planning with continuous resources in stochastic domains. *JAIR*, 2009.

[788] N. Meuleau, et al. A heuristic search approach to planning with continuous resources in stochastic domains. *JAIR*, 2009.

[789] O. Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *Internat. Jour. of Advanced Robotic Systems*, 2004.

[790] A. Micheli and A. Valentini. Synthesis of search heuristics for temporal planning via reinforcement learning. In *AAAI*, 2021.

[791] I. Miguel, et al. Flexible graphplan. In *ECAI*, 2000.

[792] J. Minguez, et al. Motion planning and obstacle avoidance. In *Handbook of Robotics*. Springer, 2008.

[793] S. Minton, et al. Commitment strategies in planning: A comparative analysis. In *IJCAI*, 1991.

[794] S. Minton, et al. Total order vs. partial order planning: Factors influencing performance. In *KR*, 1992.

[795] S. Mirchandani, et al. Ella: Exploration through learned language abstraction. In *NeurIPS*, 2021.

[796] A. Miyamae, et al. Natural policy gradient methods with parameter-based exploration for control tasks. *NeurIPS*, 2010.

[797] V. Mnih, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[798] T. M. Moerland, et al. A framework for reinforcement learning and planning. *arXiv:2006.15009*, 2020.

[799] T. M. Moerland, et al. Model-based reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 2023.

[800] T. B. Moeslund, et al. A survey of advances in vision-based human motion capture and analysis. *Computer Vision and Image Understanding*, 2006.

[801] M. D. Moffitt. On the modelling and optimization of preferences in constraint-based temporal reasoning. *AIJ*, 2011.

[802] M. D. Moffitt and M. E. Pollack. Partial constraint satisfaction of disjunctive temporal problems. In *FLAIRS*, 2005.

[803] M. Mohanan and A. Salgoankar. A survey of robotic motion planning in dynamic environments. *RAS*, 2018.

[804] M. Molineaux, et al. Goal-driven autonomy in a Navy strategy simulation. In *AAAI*, 2010.

[805] M. Montemerlo, et al. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI*, 2002.

[806] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *ML*, 1993.

[807] A. Mordoch, et al. Collaborative multi-agent planning with black-box agents by learning action models. In *ICAPS Wksp. on on Reliable Data-Driven Planning and Scheduling (RDDPS)*, 2022.

[808] A. Mordoch, et al. Learning safe numeric action models. In *AAAI*, 2023.

[809] B. Morisset and M. Ghallab. Learning how to combine sensory-motor functions into a robust behavior. *AIJ*, 2008.

[810] P. Morris, et al. Dynamic control of plans with temporal uncertainty. In *IJCAI*, 2001.

[811] P. H. Morris. Dynamic controllability and dispatchability relationships. In *Integration of AI and OR Techniques in Constraint Programming*, Apr. 2014.

[812] P. H. Morris and N. Muscettola. Temporal dynamic controllability revisited. In *AAAI*, 2005.

[813] D. R. Morrison, et al. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 2016.

[814] K. Mourão, et al. Learning STRIPS operators from noisy and incomplete observations. In *UAI*, 2012.

[815] D. Mourtzis, et al. Simulation in manufacturing: Review and challenges. *Procedia CIRP*, 2014.

[816] C. Muise, et al. Non-deterministic planning with conditional effects. In *ICAPS*, 2014.

[817] C. Muise, et al. PRP rebooted: Advancing the state of the art in FOND planning. In *AAAI*, 2024.

[818] C. J. Muise, et al. Improved non-

deterministic planning by exploiting state relevance. In *ICAPS*, 2012.

[819] R. Munos and A. W. Moore. Variable resolution discretization in optimal control. *ML*, 2002.

[820] H. Munoz-Avila and M. T. Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 2008.

[821] H. Muñoz-Avila, et al. SiN: Integrating case-based reasoning with task decomposition. In *IJCAI*, 2001.

[822] N. Muscettola, et al. Reformulating temporal plans for efficient execution. In *KR*, 1998.

[823] N. Muscettola, et al. Remote Agent: To boldly go where no AI system has gone before. *AIJ*, 1998.

[824] N. Muscettola, et al. IDEA: Planning at the core of autonomous reactive agents. In *IWPSS*, 2002.

[825] D. J. Musliner, et al. The evolution of CIRCA, a theory-based AI architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, 2008.

[826] K. L. Myers. CPEF: A continuous planning and execution framework. *AIMag*, 1999.

[827] O. Nachum, et al. Data-efficient hierarchical reinforcement learning. *NeurIPS*, 2018.

[828] A. Najar and M. Chetouani. Reinforcement learning with human advice: a survey. *Frontiers in Robotics and AI*, 2021.

[829] A. Nareyek, et al. Constraints and AI planning. *IEEE Intelligent Systems*, 2005.

[830] D. Nau, et al. GTPyhop: A hierarchical goal+task planner implemented in Python. In *HPlan*, July 2021.

[831] D. S. Nau, et al. General branch and bound, and its relation to A* and AO*. *AIJ*, 1984.

[832] D. S. Nau, et al. SHOP: Simple hierarchical ordered planner. In *IJCAI*, 1999.

[833] D. S. Nau, et al. Total-order planning with partially ordered subtasks. In *IJCAI*, Aug. 2001.

[834] H. Naveed, et al. A comprehensive overview of large language models. *arXiv:2307.06435*, 2023.

[835] B. Nebel and H. Burckert. Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra. *JACM*, 1995.

[836] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *AIJ*, July 1995.

[837] S. Nedunuri, et al. SMT-based synthesis of integrated task and motion plans from plan outlines. In *ICRA*, 2014.

[838] G. Neu and C. Szepesvári. Training parsers by inverse reinforcement learning. *ML*, 2009.

[839] G. Neu and C. Szepesvári. Apprenticeship learning using inverse reinforcement learning and gradient methods. *arXiv:1206.5264*, 2012.

[840] X. Neufeld, et al. Building a planner: A survey of planning systems used in commercial video games. *TG*, 2017.

[841] R. A. Newcombe and A. J. Davison. Live dense reconstruction with a single moving camera. In *CVPR*, 2010.

[842] A. Newell and G. Ernst. The search for generality. In *IFIP Congress*, volume 65, 1965.

[843] A. Newell and H. A. Simon. GPS, a program that simulates human thought. In E. A. Feigenbaum and J. A. Feldman, editors, *Computers and Thought*. McGraw-Hill, 1963.

[844] M. A. H. Newton, et al. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, 2007.

[845] J. M. Nez-Carranza and A. Calway. Unifying planar and point mapping in monocular SLAM. In *British Machine Vision Conf.*, 2010.

[846] A. Ng and M. Jordan. PEGASUS: a policy search method for large MDPs and POMDPs. In *UAI*, 2000.

[847] A. Y. Ng, et al. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, 1999.

[848] A. Y. Ng, et al. Algorithms for inverse reinforcement learning. In *ICML*, volume 1, 2000.

[849] N. Nguyen and S. Kambhampati. Reviving partial order planning. In *IJCAI*, 2001.

[850] A. Nicolin, et al. Agimus: a new framework for mapping manipulation motion plans to sequences of hierarchical task-based controllers. In *IEEE Internat. Symposium on System Integration*, 2020.

[851] S. Niekum. An integrated system for learning multi-step robotic tasks from unstructured demonstrations. In *AAAI Spring Symposium*, 2013.

[852] R. Nieuwenhuis, et al. Solving SAT and SAT modulo theories: From an ab-

stract Davis-Putnam-Logemann-Loveland procedure to DPLL (T). *JACM*, 2006.

[853] E. Nikolova and D. R. Karger. Route planning under uncertainty: The Canadian traveller problem. In *AAAI*, 2008.

[854] M. Nilsson, et al. EfficientIDC: A faster incremental dynamic controllability algorithm. In *ICAPS*, 2014.

[855] M. Nilsson, et al. Incremental dynamic controllability in cubic worst-case time. In *TIME*, 2014.

[856] N. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, 1980.

[857] Y. Niu, et al. Atp: Enabling fast llm serving via attention on top principal keys. *arXiv:2403.02352*, 2024.

[858] S. C. W. Ong, et al. Planning under uncertainty for robotic tasks with mixed observability. *IJRR*, 2010.

[859] OpenAI. GPT-4. Technical report, OpenAI, 2023.

[860] J. Oswald, et al. Large language models as planning domain generators. In *ICAPS*, 2024.

[861] L. Ouyang, et al. Training language models to follow instructions with human feedback. *arXiv:2203.02155*, 2022.

[862] R. Özalp, et al. A review of deep reinforcement learning algorithms and comparative results on inverted pendulum system. *Machine Learning Paradigms*, 2020.

[863] B. Paden, et al. A survey of motion planning and control techniques for self-driving urban vehicles. *TIV*, 2016.

[864] V. Pallagani, et al. Understanding the capabilities of large language models for automated planning. *arXiv:2305.16151*, 2023.

[865] V. Pallagani, et al. Plansformer tool: Demonstrating generation of symbolic plans using transformers. In *IJCAI*, 2023.

[866] V. Pallagani, et al. On the prospects of incorporating large language models (LLMs) in automated planning and scheduling (APS). In *ICAPS*, 2024.

[867] X. Pan, et al. Virtual to real reinforcement learning for autonomous driving. *arXiv:1704.03952*, 2017.

[868] A. Parisi, et al. Talm: Tool augmented language models. *arXiv:2205.12255*, 2022.

[869] G. I. Parisi, et al. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.

[870] R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In *NeurIPS*, 1998.

[871] S. Pateria, et al. Hierarchical reinforcement learning: A comprehensive survey. *CSUR*, 2021.

[872] S. Patra, et al. Deliberative acting, planning and learning with hierarchical operational models. *AIJ*, 2021.

[873] S. Patra, et al. Using online planning and acting to recover from cyberattacks on software-defined networks. In *AAAI*, volume 35, 2021.

[874] D. Patterson, et al. Carbon emissions and large neural network training. *arXiv:2104.10350*, 2021.

[875] D. Patterson, et al. The carbon footprint of machine learning training will plateau, then shrink. *IEEE Computer*, 2022.

[876] C. Paxton, et al. CoSTAR: Instructing collaborative robots with behavior trees and vision. In *ICRA*, 2017.

[877] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[878] F. Pecora, et al. A constraint-based approach for proactive, context-aware human support. *Jour. of Ambient Intelligence and Smart Environments*, 2012.

[879] E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *CI*, 1988.

[880] E. P. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *KR*, 1989.

[881] D. Pellier, et al. HDDL 2.1: Towards defining a formalism and a semantics for temporal HTN planning. *arXiv:2306.07353*, 2023.

[882] J. Penberthy and D. S. Weld. Temporal planning with continuous change. In *AAAI*, 1994.

[883] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *KR*, 1992.

[884] S. Peng, et al. Openscene: 3d scene understanding with open vocabularies. In *CVPR*, 2023.

[885] G. D. Penna, et al. Upmurphi: a tool for universal planning on PDDL+ problems. In *ICAPS*, 2009.

[886] M. Peot and D. Smith. Conditional nonlinear planning. In *AIPS*, 1992.

[887] R. F. Pereira, et al. Iterative depth-first search for FOND planning. In *ICAPS*, 2022.

[888] M. Péron, et al. Fast-tracking stationary momdps for adaptive management prob-

lems. *AAAI*, 2017.

[889] J. Peters and S. Schaal. Reinforcement learning by reward-weighted regression for operational space control. In *ICML*, 2007.

[890] J. Peters, et al. Natural actor-critic. In *ECML*, 2005.

[891] J. Peters, et al. Towards robot skill learning: From simple skills to table tennis. In *ECML PKDD*, 2013.

[892] J. L. Peterson. Petri nets. *CSUR*, 1977.

[893] C. A. Petri. *Communication with Automata*. PhD thesis, Institut für Instrumentelle Mathematik, Bonn, 1962.

[894] R. Petrick and F. Bacchus. Extending the knowledge-based approach to planning with incomplete information and sensing. In *ICAPS*, 2004.

[895] O. Pettersson. Execution monitoring in robotics: A survey. *RAS*, 2005.

[896] Q.-C. Pham, et al. Kinodynamic planning in the configuration space via admissible velocity propagation. In *RSS*, 2013.

[897] J. Pineau, et al. Policy-contingent abstraction for robust robot control. In *UAI*, 2002.

[898] J. Pineau, et al. Towards robotic assistants in nursing homes: Challenges and results. *RAS*, Mar. 2003.

[899] M. Pistore and P. Traverso. Planning as model checking for extended goals in nondeterministic domains. In *IJCAI*, 2001.

[900] M. Pistore and P. Traverso. Assumption-based composition and monitoring of web services. In *Test and Analysis of Web Services*. 2007.

[901] M. Pistore, et al. Symbolic techniques for planning with extended goals in nondeterministic domains. In *ECP*, 2001.

[902] M. Pistore, et al. Automated composition of web services by planning in asynchronous domains. In *ICAPS*, 2005.

[903] M. Pistore, et al. A minimalist approach to semantic annotations for web processes compositions. In *ESWC*, 2006.

[904] L. R. Planken. Incrementally solving the STP by enforcing partial path consistency. In *PLANSIG*, 2008.

[905] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[906] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP*, 1989.

[907] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *FOCS*, 1990.

[908] I. Pohl. Heuristic search viewed as path finding in a graph. *AIJ*, 1970.

[909] M. E. Pollack and J. F. Horty. There's more to life than making plans: Plan management in dynamic, multiagent environments. *AIMag*, 1999.

[910] A. S. Polydoros and L. Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *JIRS*, 2017.

[911] F. Pommereau. *Algebras of coloured Petri nets*. Lambert Academic Publishing, 2010.

[912] F. Pommerening, et al. From non-negative to general operator cost partitioning. In *AAAI*, 2015.

[913] J. Porteous, et al. On the extraction, ordering, and usage of landmarks in planning. In *ECP*, 2001.

[914] J. Powell, et al. Active and interactive discovery of goal selection knowledge. In *FLAIRS*, 2011.

[915] E. Prassler and K. Kosuge. Domestic robotics. In *Handbook of Robotics*. Springer, 2008.

[916] S. Prentice and N. Roy. The belief roadmap: Efficient planning in belief space by factoring the covariance. *IJRR*, 2009.

[917] L. Pryor and G. Collins. Planning for contingency: A decision based approach. *JAIR*, 1996.

[918] M. Pternea, et al. The RL/LLM taxonomy tree: Reviewing synergies between reinforcement learning and large language models. *JAIR*, 2024.

[919] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[920] F. Py, et al. A systematic agent framework for situated autonomous systems. In *AAMAS*, 2010.

[921] D. V. Pynadath and M. P. Wellman. Probabilistic state-dependent grammars for plan recognition. In *UAI*, 2000.

[922] C. R. Qi, et al. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *NeurIPS*, 2017.

[923] W. Qiu and H. Zhu. Programmatic reinforcement learning without oracles. In *ICLR*, 2021.

[924] B. Quartey, et al. Exploiting contextual structure to generate useful auxiliary tasks. *arXiv:2303.05038*, 2023.

[925] R. Quiniou, et al. Application of ILP to cardiac arrhythmia characterization for

chronicle recognition. In *ILP*. 2001.

[926] G. Rabideau, et al. Iterative repair planning for spacecraft operations in the ASPEN system. In *i-SAIRAS*, 1999.

[927] A. Radford, et al. Learning transferable visual models from natural language supervision. In *ICML*, 2021.

[928] A. N. Raghavan, et al. Bidirectional online probabilistic planning. In *ICAPS*, 2012.

[929] K. Rajan and F. Py. T-REX: Partitioned inference for AUV mission control. In *Further Advances in Unmanned Marine Vehicles*. 2012.

[930] K. Rajan and A. Saffiotti, editors. *Special Issue on AI and Robotics*. AIJ, 2017.

[931] K. Rajan, et al. Towards deliberative control in marine robotics. In *Marine Robot Autonomy*. 2012.

[932] A. Rajvanshi, et al. Saynav: Grounding large language models for dynamic planning to navigation in new environments. In *ICAPS*, 2024.

[933] M. Ramirez and H. Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *AAAI*, 2010.

[934] M. Ramírez and S. Sardina. Directed fixed-point regression-based planning for non-deterministic domains. In *ICAPS*, 2014.

[935] M. Ramírez, et al. Behavior composition as fully observable non-deterministic planning. In *ICAPS*, 2013.

[936] J. Ramon, et al. Transfer learning in reinforcement learning problems through partial policy recycling. In *ECML*, 2007.

[937] D. Rao, et al. Performance of the RDDL planners. In *ICOACS*, 2016.

[938] H. Ravichandar, et al. Recent advances in robot learning from demonstration. *ARCRAS*, 2020.

[939] S. Reddy, et al. Sqil: Imitation learning via reinforcement learning with sparse rewards. *arXiv:1905.11108*, 2019.

[940] J. H. Reif. Complexity of the mover's problem and generalizations. In *IEEE Symposium on Foundations of Computer Science (SFCS)*, 1979.

[941] S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 2010.

[942] S. Richter, et al. Landmarks revisited. In *AAAI*, volume 8, 2008.

[943] M. Riedmiller, et al. Reinforcement learning for robot soccer. *Autonomous Robots*, 2009.

[944] J. Rintanen. Constructing conditional plans by a theorem-prover. *JAIR*, 1999.

[945] J. Rintanen. An iterative algorithm for synthesizing invariants. In *AAAI*, 2000.

[946] J. Rintanen. Backward plan construction for planning as search in belief space. In *AIPS*, 2002.

[947] J. Rintanen. Conditional planning in the discrete belief space. In *IJCAI*, 2005.

[948] J. Rintanen. Planning as satisfiability: Heuristics. *AIJ*, 2012.

[949] J. Rintanen. Madagascar: Scalable planning with SAT. *Internat. Planning Competition*, 2014.

[950] D. Robert, et al. Learning multi-view aggregation in the wild for large-scale 3d semantic segmentation. In *CVPR*, 2022.

[951] C. Rodrigues, et al. Incremental learning of relational action models in noisy environments. In *ILP*, 2010.

[952] C. Rodrigues, et al. Incremental learning of relational action rules. In *ICMLA*, 2010.

[953] C. Rodrigues, et al. Active learning of relational action models. In *ILP*, 2011.

[954] I. D. Rodriguez, et al. Learning first-order representations for planning from black box states: New results. In *KR*, 2021.

[955] I. D. Rodriguez, et al. Flexible FOND planning with explicit fairness assumptions. *JAIR*, 2022.

[956] M. D. Rodriguez-Moreno, et al. IPSS: A hybrid approach to planning and scheduling integration. *TDKE*, 2006.

[957] G. Röger and M. Helmert. The more, the merrier: Combining heuristic estimators for satisficing planning. In *ICAPS*, 2010.

[958] G. Röger, et al. Optimal planning in the presence of conditional effects: Extending LM-Cut with context-splitting. In *ECAI*, 2014.

[959] D. M. Roijers and S. Whiteson. *Multi-Objective Decision Making*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2017.

[960] S. Ross and J. Pineau. Model-based bayesian reinforcement learning in large structured domains. In *UAI*, 2008.

[961] S. Ross, et al. Online planning algorithms for POMDPs. *JAIR*, 2008.

[962] N. Rossetti, et al. Learning general policies for planning through GPT models. In *ICAPS*, 2024.

[963] N. Rudin, et al. Advanced skills by learning locomotion and local navigation end-

to-end. *arXiv:2209.12827*, 2022.

[964] G. A. Rummery and M. Niranjan. Online Q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, 1994.

[965] S. Russell. Learning agents for uncertain environments. In *COLT*, 1998.

[966] S. Russell. *Human compatible: AI and the problem of control*. Penguin, 2019.

[967] S. Russell and P. Norvig. *AIMA (4th Edition)*. Pearson, 2020.

[968] R. Sabbadin. A possibilistic model for qualitative sequential decision problems under uncertainty in partially observable environments. *arXiv:1301.6736*, 2013.

[969] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *AIJ*, 1974.

[970] E. Sacerdoti. The nonlinear nature of plans. In *IJCAI*, 1975.

[971] E. Sacerdoti. *A Structure for Plans and Behavior*. American Elsevier, 1977.

[972] M. Samadi, et al. Learning from multiple heuristics. In *AAAI*, 2008.

[973] M. Samadi, et al. Using the Web to interactively learn to find objects. In *AAAI*, 2012.

[974] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kauffmann, 2006.

[975] S. Samsi, et al. From words to watts: Benchmarking the energy costs of large language model inference. In *IEEE High Performance Extreme Computing Conference*, 2023.

[976] A. L. Samuel. Some studies in machine learning using the game of checkers: II—recent progress. *IBM Jour. of Research and Development*, 1959.

[977] E. Sandewall. *Features and Fluents: The Representation of Knowledge about Dynamical Systems*. Oxford Univ. Press, 1994.

[978] E. Sandewall and R. Rönnquist. A representation of action structures. In *AAAI*, 1986.

[979] S. Sanner. Relational dynamic influence diagram language (RDDL): Language description. Technical report, NICTA, 2010.

[980] P. H. R. Q. A. Santana and B. C. Williams. Chance-constrained consistency for probabilistic temporal plan networks. In *ICAPS*, Nov. 2014.

[981] S. Sardiña, et al. Hierarchical planning in BDI agent programming languages: A formal approach. In *AAMAS*, May 2006.

[982] E. Scala and A. Grastien. Nondeterministic conformant planning using a counterexample-guided incremental compilation to classical planning. In *ICAPS*, 2021.

[983] E. Scala, et al. Landmarks for numeric planning problems. In *IJCAI*, 2017.

[984] T. Schaul, et al. Universal value function approximators. In *ICML*, 2015.

[985] S. Scheck, et al. Knowledge compilation for nondeterministic action languages. In *ICAPS*, 2021.

[986] E. Scheide, et al. Behavior tree learning for robotic task planning through monte carlo DAG search over a formal grammar. In *ICRA*, 2021.

[987] B. Scherrer and B. Lesner. On the use of non-stationary policies for stationary infinite-horizon Markov decision processes. In *NeurIPS*, 2012.

[988] T. Schick, et al. Toolformer: Language models can teach themselves to use tools. *arXiv:2302.04761*, 2023.

[989] D. Schreiber. Lilotane: A lifted SAT-based approach to hierarchical planning. *JAIR*, 2021.

[990] J. Schulman, et al. Trust region policy optimization. In *ICML*, 2015.

[991] J. Schulman, et al. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[992] D. G. Schultz and J. L. Melsa. *State Functions and Linear Control Systems*. McGraw-Hill, 1967.

[993] J. T. Schwartz and M. Sharir. On the "piano movers" problem. general techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Mathematics*, 1983.

[994] J. Seipp, et al. Saturated cost partitioning for optimal classical planning. *JAIR*, 2020.

[995] L. Serafini and A. d. Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *arXiv:1606.04422*, 2016.

[996] L. Serafini and P. Traverso. Learning abstract planning domains and mappings to real world perceptions. In *AIxIA*, volume 11946, 2019.

[997] S. Shah, et al. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2018.

[998] G. Shani, et al. A survey of point-based POMDP solvers. *JAAMAS*, 2012.

[999] C. E. Shannon. Programming a computer for playing chess. *Philosophical Magazine and Jour. of Science*, 1950.

[1000] C. E. Shannon. Presentation of a maze-solving machine. In *Cybernetics, Transactions of the Eighth Conf.*, 1951.

[1001] C. E. Shannon. Prediction and entropy of printed english. *Bell System Technical Jour.*, 1951.

[1002] D. Shaparau, et al. Contingent planning with goal preferences. In *AAAI*, 2006.

[1003] D. Shaparau, et al. Fusing procedural and declarative planning goals for nondeterministic domains. In *AAAI*, 2008.

[1004] A. Sharma, et al. Dynamics-aware unsupervised discovery of skills. *arXiv:1907.01657*, 2019.

[1005] P. Sharma, et al. Skill induction and planning with latent language. *arXiv:2110.01517*, 2021.

[1006] W. Shen, et al. Learning domain-independent planning heuristics with hypergraph networks. In *ICAPS*, 2020.

[1007] H. Shi, et al. Continual learning of large language models: A comprehensive survey. *arXiv:2404.16789*, 2024.

[1008] N. Shinn, et al. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv:2303.11366*, 2023.

[1009] V. Shivashankar, et al. A hierarchical goal-based formalism and algorithm for single-agent planning. In *AAMAS*, 2012.

[1010] V. Shivashankar, et al. The GoDeL planning system: A more perfect union of domain-independent and hierarchical planning. In *IJCAI*, 2013.

[1011] Y. Shoahm and D. McDermott. Problems in formal temporal reasoning. *AIJ*, 1988.

[1012] Y. Shoham. Temporal logic in AI: semantical and ontological considerations. *AIJ*, 1987.

[1013] L. Shtutland, et al. Unavoidable deadends in deterministic partially observable contingent planning. *JAAMAS*, 2023.

[1014] B. Siciliano and O. Khatib, editors. *The Handbook of Robotics*. Springer, 2008.

[1015] D. Silver and J. Veness. Monte-Carlo planning in large POMDPs. In *NeurIPS*, 2010.

[1016] D. Silver, et al. Learning from demonstration for autonomous navigation in complex unstructured terrain. *IJRR*, 2010.

[1017] D. Silver, et al. Deterministic policy gradient algorithms. In *ICML*, 2014.

[1018] D. Silver, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[1019] D. Silver, et al. Mastering the game of go without human knowledge. *Nature*, 2017.

[1020] D. Silver, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 2018.

[1021] T. Silver, et al. Generalized planning in PDDL domains with pretrained large language models. *arXiv:2305.11014*, 2023.

[1022] T. Siméon, et al. Visibility-based probabilistic roadmaps for motion planning. *Advanced Robotics*, 2000.

[1023] T. Siméon, et al. Manipulation planning with probabilistic roadmaps. *IJRR*, 2004.

[1024] R. Simmons. Concurrent planning and execution for autonomous robots. *IEEE Control Systems*, 1992.

[1025] R. Simmons. Structured control for autonomous robots. *TRA*, 1994.

[1026] R. Simmons and D. Apfelbaum. A task description language for robot control. In *IROS*, 1998.

[1027] R. Simmons and R. Davis. Generate, test and debug: Combining associational rules and causal models. In *IJCAI*, 1987.

[1028] C. Simpkins, et al. Towards adaptive programming: integrating reinforcement learning into a programming language. In *ACM Object-oriented programming systems languages and applications*, 2008.

[1029] I. Singh, et al. Progprompt: program generation for situated robot task planning using large language models. *Autonomous Robots*, 2023.

[1030] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *ML*, 1996.

[1031] E. Sirin, et al. HTN planning for Web service composition using SHOP2. *Jour. of Web Semantics*, 2004.

[1032] B. F. Skinner. *The Behavior of Organisms: An Experimental Analysis*. Appleton, 1938.

[1033] D. E. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *IJCAI*, 1999.

[1034] D. E. Smith and D. S. Weld. Conformant Graphplan. In *AAAI*, 1998.

[1035] D. E. Smith and D. S. Weld. Temporal planning with mutual exclusion reasoning. In *IJCAI*, 1999.

[1036] D. E. Smith, et al. Bridging the gap between planning and scheduling. *KER*, 2000.

[1037] D. E. Smith, et al. The ANML language. In *KEPS*, 2008.

[1038] L. Smith, et al. A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv:2208.07860*, 2022.

[1039] S. J. J. Smith, et al. Integrating electrical and mechanical design and process planning. In *Knowledge Intensive CAD*. 1997.

[1040] S. J. J. Smith, et al. Computer bridge: A big win for AI planning. *AIMag*, 1998.

[1041] T. Smith and R. Simmons. Heuristic search value iteration for POMDPs. In *UAI*, 2004.

[1042] S. Sohrabi and S. A. McIlraith. Preference-based web service composition: A middle ground between execution and search. In *ISWC*. 2010.

[1043] S. Sohrabi, et al. HTN planning with preferences. In *IJCAI*, 2009.

[1044] S. Sohrabi, et al. HTN planning for the composition of stream processing applications. In *ICAPS*, 2013.

[1045] A. Somani, et al. DESPOT: Online POMDP planning with regularization. *NeurIPS*, 2013.

[1046] J. Song, et al. Self-refined large language model as automated reward function designer for deep reinforcement learning in robotics. *arXiv:2309.06687*, 2023.

[1047] C. I. Sprague, et al. Improving the modularity of AUV control systems using behaviour trees. In *IEEE/OES Autonomous Underwater Vehicle Workshop*, 2018.

[1048] M. Sridharan, et al. HiPPo: Hierarchical POMDPs for planning information processing and sensing actions on a robot. In *ICAPS*, 2008.

[1049] B. Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI*, 2000.

[1050] N. Srivastava, et al. Dropout: a simple way to prevent neural networks from overfitting. *JMLR*, 2014.

[1051] C. Stachniss and W. Burgard. Exploration with active loop-closing for FastSLAM. In *IROS*, 2004.

[1052] S. Ståhlberg, et al. Learning generalized policies without supervision using GNNs. In *KR*, 2022.

[1053] S. Ståhlberg, et al. Learning general policies with policy gradient methods. In *KR*, 2023.

[1054] O. Stasse, et al. TALOS: A new humanoid research platform targeted for industrial applications. In *Humanoids*, 2017.

[1055] J. Stedl and B. Williams. A fast incremental dynamic controllability algorithm. In *ICAPS Wksp. on Plan Execution*, 2005.

[1056] M. Steinmetz, et al. Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art. *JAIR*, 2016.

[1057] R. Stern and B. Juba. Efficient, safe, and probably approximately complete learning of action models. In *IJCAI*, 2017.

[1058] S. Stock, et al. Hierarchical hybrid planning in a mobile service robot. In *KI*, 2015.

[1059] S. Stock, et al. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *IROS*, 2015.

[1060] A. L. Strehl, et al. Efficient structure learning in factored-state mdps. In *AAAI*, volume 7, 2007.

[1061] J. Styrud, et al. Combining planning and learning of behavior trees for robotic assembly. In *ICRA*, 2022.

[1062] Z. Su, et al. Learning manipulation graphs from demonstrations using multimodal sensory signals. In *ICRA*, 2018.

[1063] S. Subramanian, et al. Reclip: A strong zero-shot baseline for referring expression comprehension. *arXiv:2204.05991*, 2022.

[1064] Z. Sun, et al. Human action recognition from various data modalities: A review. *PAMI*, 2022.

[1065] Z. Sunberg and M. Kochenderfer. Online algorithms for pomdps with continuous state, action, and observation spaces. In *ICAPS*, 2018.

[1066] M. Suomalainen, et al. A survey of robot manipulation in contact. *RAS*, 2022.

[1067] H. Surmann, et al. An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments. *RAS*, 2003.

[1068] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984.

[1069] R. S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 1991.

[1070] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[1071] R. S. Sutton, et al. Policy gradient methods for reinforcement learning with function approximation. *NeurIPS*, 1999.

[1072] C. Szepesvári and W. D. Smart. Interpolation-based Q-learning. In *ICML*, 2004.

[1073] I. Szita and A. Lörincz. Learning Tetris using the noisy cross-entropy method. *Neural Computation*, 2006.

[1074] P. Tadepalli, et al. Relational reinforcement learning: An overview. In *ICML Workshop on Relational Reinforcement Learning*, 2004.

[1075] H. A. Taha. *Integer Programming: Theory, Applications, and Computations*. Academic Press, 1975.

[1076] C. Tang, et al. GraspGPT: Leveraging semantic knowledge from a large language model for task-oriented grasping. *arXiv:2307.13204*, 2023.

[1077] D. Tanneberg and M. Gienger. Learning type-generalized actions for symbolic planning. In *IROS*, 2023.

[1078] R. E. Tarjan. Depth-first search and linear graph algorithms. *SICOMP*, 1972.

[1079] A. Tate. Generating project networks. In *IJCAI*, 1977.

[1080] A. Tate, et al. *O-Plan2: An Architecture for Command, Planning and Control*. Morgan-Kaufmann, 1994.

[1081] H. Täubig, et al. Medical robotics and computer-integrated surgery. In *Handbook of Robotics*. Springer, 2008.

[1082] M. E. Taylor and P. Stone. Transfer learning for reinforcement learning domains: A survey. *JMLR*, 2009.

[1083] M. E. Taylor, et al. Transfer learning via inter-task mappings for temporal difference learning. *JMLR*, 2007.

[1084] Y. Teh, et al. Distral: Robust multitask reinforcement learning. *NeurIPS*, 2017.

[1085] F. Teichteil-Königsbuch. Fast incremental policy compilation from plans in hybrid probabilistic domains. In *ICAPS*, 2012.

[1086] F. Teichteil-Königsbuch. Stochastic safest and shortest path problems. In *AAAI*, 2012.

[1087] F. Teichteil-Königsbuch, et al. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *ICAPS*, 2008.

[1088] F. Teichteil-Königsbuch, et al. Incremental plan aggregation for generating policies in MDPs. In *AAMAS*, 2010.

[1089] F. Teichteil-Königsbuch, et al. Extending classical planning heuristics to probabilistic planning with dead-ends. In *AAAI*, 2011.

[1090] S. Teng, et al. Motion planning for autonomous driving: The state of the art and future perspectives. *TIV*, 2023.

[1091] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 1994.

[1092] G. Tesauro et al. Temporal difference learning and TD-Gammon. *CACM*, 1995.

[1093] J. T. Thayer, et al. Learning inadmissible heuristics during search. In *ICAPS*, 2011.

[1094] N. C. Thompson, et al. The computational limits of deep learning. *arXiv:2007.05558*, 2022.

[1095] E. L. Thorndike. *Animal Intelligence*. Macmillan, 1911.

[1096] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium*. Morgan Kaufmann, 2002.

[1097] S. Thrun. Stanley: The robot that won the DARPA grand challenge. *JFR*, 2006.

[1098] E. Todorov, et al. Mujoco: A physics engine for model-based control. In *IROS*, 2012.

[1099] D. Tola and P. Corke. Understanding urdf: A survey based on user experience. *arXiv:2302.13442*, 2023.

[1100] L. Torrey and J. Shavlik. Transfer learning. In *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*. IGI global, 2010.

[1101] M. Toussaint. Logic-geometric programming - an optimization-based approach to combined task and motion planning. In *IJCAI*, 2015.

[1102] F. W. Trevizan, et al. Heuristic search in dual space for constrained stochastic shortest path problems. In *ICAPS*, 2016.

[1103] F. W. Trevizan, et al. Efficient solutions for stochastic shortest path problems with dead ends. In *UAI*, 2017.

[1104] F. W. Trevizan, et al. Occupation measure heuristics for probabilistic planning. In *ICAPS*, 2017.

[1105] B. Triggs, et al. Bundle adjustment – a modern synthesis. In B. Triggs, et al., editors, *Internat. Wksp. on Vision Algorithms*, 1999.

[1106] A. Trott, et al. Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards. *NeurIPS*, 2019.

[1107] P. Tuffield and H. Elias. The Shadow robot mimics human actions. *Industrial Robot: An International Jour.*, 2003.

[1108] A. Turing. Computing machinery and intelligence. *Mind*, 1950.

[1109] UN AI Advisory Board. Governing AI for

Humanity, Interim Report, 2023.

[1110] F. W. Vaandrager. Active automata learning: from l^* to $l^{\#}$. In *FMCAD*, 2021.

[1111] K. Valmeekam, et al. On the planning abilities of large language models (a critical investigation with a proposed benchmark). *arXiv:2302.06706*, 2023.

[1112] G. M. Van de Ven, et al. Brain-inspired replay for continual learning with artificial neural networks. *Nature Communications*, 2020.

[1113] M. van den Briel, et al. Loosely coupled formulations for automated planning: An integer programming perspective. *JAIR*, 2008.

[1114] R. Van Der Krogt and M. De Weerdt. Plan repair as an extension of planning. In *ICAPS*, 2005.

[1115] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In *CAV*, 1995.

[1116] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, 1995.

[1117] M. Y. Vardi. From verification to synthesis. In *Internat. Conf. on Verified Software: Theories, Tools, Experiments*, 2008.

[1118] J. W. Vásquez, et al. Enhanced chronicle learning for process supervision. *IFAC*, 2017.

[1119] A. Vaswani, et al. Attention is all you need. *NeurIPS*, 2017.

[1120] S. Vattam, et al. Breadth of approaches to goal reasoning: A research survey. In *ACS Wksp. on Goal Reasoning*, 2013.

[1121] M. Vecerik, et al. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *Computing Research Repository*, 2017.

[1122] J. Velez, et al. Planning to perceive: Exploiting mobility for robust object detection. In *ICAPS*, 2011.

[1123] M. Veloso and P. Stone. FLECS: planning with a flexible commitment strategy. *JAIR*, 1995.

[1124] M. M. Veloso and J. Carbonell. Derivational analogy in PRODIGY: Automating case acquisition, storage and utilization. *ML*, 1993.

[1125] M. M. Veloso and P. Rizzo. Mapping planning actions and partially-ordered plans into execution knowledge. In *Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*, 1998.

[1126] M. M. Veloso, et al. Integrating planning and learning: The PRODIGY architecture. *JETAI*, 1995.

[1127] S. Vemprala, et al. ChatGPT for robotics: Design principles and model abilities. *Microsoft Autonomous Systems and Robotics Research*, 2023.

[1128] S. Vere. Planning in time: Windows and duration for activities and goals. *PAMI*, 1983.

[1129] G. Verfaillie, et al. How to model planning and scheduling problems using constraint networks on timelines. *KER*, 2010.

[1130] C. K. Verginis, et al. KDF: Kinodynamic motion planning via geometric sampling-based algorithms and funnel control. *T-RO*, 2023.

[1131] A. Verma, et al. Programmatically interpretable reinforcement learning. In *ICML*, 2018.

[1132] A. Verma, et al. Imitation-projected programmatic reinforcement learning. *NeurIPS*, 2019.

[1133] A. Verma et al. *Programmatic reinforcement learning*. PhD thesis, Univ. of Texas at Austin, 2021.

[1134] M. Verma, et al. Preference proxies: Evaluating large language models in capturing human preferences in human-ai tasks. In *ICML Workshop on The Many Facets of Preference-Based Learning*, 2023.

[1135] P. Verma, et al. Automatic generation of behavior trees for the execution of robotic manipulation tasks. In *ETFA*, 2021.

[1136] V. Verma, et al. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *i-SAIRAS*, 2005.

[1137] T. Verweij. A hierarchically-layered multiplayer bot system for a first-person shooter. Master's thesis, Vrije Universiteit of Amsterdam, 2007.

[1138] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *JETAI*, 1999.

[1139] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraints networks dedicated to planning. In *ECAI*, 1996.

[1140] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI*, 1986.

[1141] M. Vilain, et al. Constraint propagation algorithms for temporal reasoning: a revised report. In *Readings in Qualitative Reasoning about Physical Systems*. 1989.

[1142] O. Vinyals, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019.

[1143] T. Vodopivec, et al. On monte carlo tree search and reinforcement learning. *JAIR*, 2017.

[1144] V.-T. Vu, et al. Automatic video interpretation: A novel algorithm for temporal scenario recognition. In *IJCAI*, 2003.

[1145] R. Waldinger. Achieving several goals simultaneously. In *Machine Intelligence*. 1977.

[1146] T. Walsh, et al. Integrating sample-based planning and model-based reinforcement learning. In *AAAI*, 2010.

[1147] C. Wang, et al. Large language models for multi-modal human-robot interaction. *arXiv:2401.15174*, 2024.

[1148] F. Y. Wang, et al. A Petri-net coordination model for an intelligent mobile robot. *SMC*, 1991.

[1149] G. Wang, et al. Voyager: An open-ended embodied agent with large language models. *arXiv:2305.16291v*, 2023.

[1150] L. Wang, et al. A survey on large language model based autonomous agents. *arXiv:2308.11432*, 2023.

[1151] L. Wang, et al. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv:2305.04091*, 2023.

[1152] T. Wang, et al. Nervenet: Learning structured policy with graph neural networks. In *ICLR*, 2018.

[1153] X. Wang. Planning while learning operators. In *AAAI*, 1996.

[1154] Z. Wang, et al. Incremental reinforcement learning with prioritized sweeping for dynamic environments. *TMECH*, 2019.

[1155] I. Warfield, et al. Adaptation of hierarchical task network plans. In *FLAIRS*, 2007.

[1156] G. Warnell, et al. Deep TAMER: Interactive agent shaping in high-dimensional state spaces. In *AAAI*, 2018.

[1157] D. H. D. Warren. Generating conditional plans and programs. In *Summer Conf. on Artificial Intelligence and Simulation of Behaviour*, 1976.

[1158] C. Watkins and P. Dayan. Q-learning. *ML*, 1992.

[1159] O. Watkins, et al. Teachable reinforcement learning via advice distillation. In *NeurIPS*, 2021.

[1160] J. Wei, et al. Emergent abilities of large language models. *arXiv:2206.07682*, 2022.

[1161] M. Weiring and M. Otterlo. *Reinforcement Learning*. Springer, 2012.

[1162] D. Weld. Recent advances in AI planning. *AIMag*, 1999.

[1163] D. S. Weld. An introduction to least commitment planning. *AIMag*, 1994.

[1164] D. S. Weld and O. Etzioni. The first law of robotics (a call to arms). In *AAAI*, 1994.

[1165] D. S. Weld, et al. Extending Graphplan to handle uncertainty and sensing actions. In *AAAI*, 1998.

[1166] A. M. Wells, et al. Learning feasibility for task and motion planning in tabletop environments. *IEEE Robotics and Automation Letters*, 2019.

[1167] P. Werbos. Advanced forecasting methods for global crisis warning and models of intelligence. *General System Yearbook*, 1977.

[1168] D. J. White. Multi-objective infinite-horizon discounted markov decision processes. *Journal of Mathematical Analysis and Applications*, 1982.

[1169] D. Wilkins. Recovering from execution errors in SIPE. *CI*, 1985.

[1170] D. Wilkins and M. desJardins. A call for knowledge-based planning. *AIMag*, 2001.

[1171] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

[1172] D. E. Wilkins. Can AI planners solve practical problems? *CI*, 1990.

[1173] D. E. Wilkins and K. L. Myers. A common knowledge representation for plan generation and reactive execution. *Jour. of Logic and Computation*, 1995.

[1174] B. C. Williams and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*, 2001.

[1175] B. C. Williams and P. P. Nayak. A model-based approach to reactive self-configuring systems. In *AAAI*, 1996.

[1176] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *ML*, 1992.

[1177] A. Wilson, et al. Multi-task reinforcement learning: a hierarchical bayesian approach. In *ICML*, 2007.

[1178] M. Wilson and D. W. Aha. A goal reasoning model for autonomous underwater vehicles. In *Proc. 9th Goal Reasoning*

Workshop, 2021.

[1179] C. M. Wilt and W. Ruml. When does weighted A* fail? In *SOCS*. 2012.

[1180] C. M. Wilt and W. Ruml. Building a heuristic for greedy search. In *SOCS*, 2015.

[1181] D. Wingate and K. D. Seppi. Prioritization methods for accelerating MDP solvers. *JMLR*, 2005.

[1182] Y. Wu and T. S. Huang. Vision-based gesture recognition: A review. In *Gesture-Based Communication in Human-Computer Interaction*. 1999.

[1183] K. Xi, et al. Neuro-symbolic learning of lifted action models from visual traces. In *ICAPS*, 2024.

[1184] S. Xiao, et al. Model-guided synthesis for LTL over finite traces. In *VMCAI*, 2024.

[1185] X. Xiao, et al. Motion planning and control for mobile robot navigation using machine learning: a survey. *Autonomous Robots*, 2022.

[1186] Z. Xiao, et al. Refining HTN methods via task insertion with preferences. In *AAAI*, 2020.

[1187] F. Xie, et al. Understanding and improving local exploration for GBFS. In *ICAPS*, 2015.

[1188] T. Xie, et al. Text2reward: Automated dense reward function generation for reinforcement learning. *arXiv:2309.11489*, 2023.

[1189] Y. Xie, et al. Translating natural language to planning goals with large-language models. *arXiv:2302.05128*, 2023.

[1190] H. Xiong, et al. Deterministic policy gradient: Convergence analysis. *UAI*, 2022.

[1191] J. Z. Xu and J. E. Laird. Instance-based online learning of deterministic relational action models. In *AAAI*, 2010.

[1192] J. Z. Xu and J. E. Laird. Combining learned discrete and continuous action models. In *AAAI*, 2011.

[1193] J. Z. Xu and J. E. Laird. Learning integrated symbolic and continuous action models for continuous domains. In *AAAI*, 2013.

[1194] L. Xu, et al. Accelerating integrated task and motion planning with neural feasibility checking. *arXiv:2203.10568*, 2022.

[1195] M. Xu, et al. A simple baseline for open-vocabulary semantic segmentation with pre-trained vision-language model. In *ECCV*, 2022.

[1196] Y. Xu, et al. Discriminative learning of beam-search heuristics for planning. In *IJCAI*, 2007.

[1197] K. Yang, et al. If LLM is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. *arXiv:2401.00812*, 2024.

[1198] Q. Yang. Formalizing planning knowledge for hierarchical planning. *CI*, 1990.

[1199] Q. Yang. *Intelligent Planning: A Decomposition and Abstraction Based Approach*. Springer, 1997.

[1200] Q. Yang, et al. Learning action models from plan examples using weighted MAX-SAT. *AIJ*, 2007.

[1201] Z. Yang, et al. Hierarchical deep reinforcement learning for continuous action control. *IEEE Transactions on Neural Networks and Learning Systems*, 2018.

[1202] Z. Yang, et al. Sequence-based plan feasibility prediction for efficient task and motion planning. *arXiv:2211.01576*, 2022.

[1203] S. Yao, et al. Tree of thoughts: Deliberate problem solving with large language models. *arXiv:2305.10601*, 2023.

[1204] Y. Yao, et al. Intention progression using quantitative summary information. In *AAMAS*, 2021.

[1205] Z. Yi, et al. The dynamic anchoring agent: A probabilistic object anchoring framework for semantic world modeling. In *FLAIRS*, 2024.

[1206] S. Yoon and S. Kambhampati. Towards model-lite planning: A proposal for learning & planning with incomplete domain models. In *ICAPS-07 Workshop on AI Planning and Learning*, 2007.

[1207] S. Yoon, et al. Learning heuristic functions from relaxed plans. In *ICAPS*, 2006.

[1208] S. Yoon, et al. FF-Replan: A baseline for probabilistic planning. In *ICAPS*, 2007.

[1209] S. Yoon, et al. Probabilistic planning via determinization in hindsight. In *AAAI*, 2008.

[1210] S. W. Yoon, et al. Learning control knowledge for forward search planning. *JMLR*, 2008.

[1211] K. Yoshida and B. Wilcox. Space robots and systems. In *Handbook of Robotics*. Springer, 2008.

[1212] H. Younes and M. Littman. PPDDL: The probabilistic planning domain definition language. Technical report, Carnegie Mellon University, 2004.

[1213] H. Younes and R. Simmons. On the role of ground actions in refinement planning. In *AIPS*, 2002.

[1214] H. Younes and R. Simmons. VHPOP: Versatile heuristic partial order planner. *JAIR*, 2003.

[1215] H. Younes and R. Simmons. Solving generalized semi-Markov decision processes using continuous phase-type distributions. In *AAAI*, 2004.

[1216] H. L. Younes and R. G. Simmons. Solving generalized semi-Markov decision processes using continuous phase-type distributions. In *AAAI*, 2004.

[1217] E. Yurtsever, et al. A survey of autonomous driving: Common practices and emerging technologies. *IEEE Access*, 2020.

[1218] P. Zaidins, et al. Implicit dependency detection for HTN plan repair. In *HPlan*, 2023.

[1219] V. Zambaldi, et al. Relational deep reinforcement learning. *arXiv:1806.01830*, 2018.

[1220] A. Zela, et al. Understanding and robustifying differentiable architecture search. *arXiv:1909.09656*, 2019.

[1221] C. Zhang, et al. Large language models for human-robot interaction: A review. *Biomimetic Intelligence and Robotics*, 2023.

[1222] H. Zhang, et al. Building cooperative embodied agents modularly with large language models. *arXiv:2307.02485*, 2023.

[1223] J. Zhang, et al. Vision-language models for vision tasks: A survey. *arXiv:2304.00685*, 2024.

[1224] R. Zhang, et al. Leveraging human guidance for deep reinforcement learning tasks. *arXiv:1909.09906*, 2019.

[1225] W. Zhang and T. G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, 1995.

[1226] Y. Zhang, et al. Efficient reinforcement learning from demonstration via bayesian network-based knowledge extraction. *Computational Intelligence and Neuroscience*, 2021.

[1227] P. Zhao, et al. An in-depth survey of large language model-based artificial intelligence agents. *arXiv:2309.14365*, 2023.

[1228] W. Zhao, et al. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *IEEE Symposium Series on Computational Intelligence*, 2020.

[1229] W. X. Zhao, et al. A survey of large language models. *arXiv:2303.18223*, 2023.

[1230] S. Zhu and G. D. Giacomo. Synthesis of maximally permissive strategies for LTLf specifications. In *IJCAI*, 2022.

[1231] Z. Zhu, et al. Transfer learning in deep reinforcement learning: A survey. *PAMI*, 2023.

[1232] H. H. Zhuo, et al. Learning complex action models with quantifiers and logical implications. *AIJ*, 2010.

[1233] H. H. Zhuo, et al. Refining incomplete planning domain models through plan traces. In *IJCAI*, 2013.

[1234] H. H. Zhuo, et al. Learning hierarchical task network domains from partially observed plan traces. *AIJ*, 2014.

[1235] B. D. Ziebart, et al. Modeling interaction via the principle of maximum causal entropy. In *ICML*, 2010.

[1236] T. Zimmerman and S. Kambhampati. Learning-assisted automated planning: looking back, taking stock, going forward. *AIMAG*, 2003.

[1237] V. A. Ziparo, et al. Petri net plans. *JAAMAS*, 2011.

Index