Implementation of two Algorithms for the Threshold Synthesis Problem

Christian Schilling
Institut für Informatik
Universität Freiburg
Germany
schillic@informatik.uni-freiburg.de

Jan-Georg Smaus
IRIT
Université de Toulouse
France
smaus@irit.fr

Fabian Wenzelmann
Institut für Informatik
Universität Freiburg
Germany
wenzelmf@informatik.uni-freiburg.de

ABSTRACT

A *linear pseudo-Boolean constraint* (LPB) is an expression of the form $a_1 \cdot \ell_1 + \ldots + a_m \cdot \ell_m \geq d$, where each ℓ_i is a *literal* (it assumes the value 1 or 0 depending on whether a propositional variable x_i is true or false) and $a_1, \ldots, a_m, d$ are natural numbers. An LPB represents a Boolean function, and those Boolean functions that can be represented by exactly one LPB are called *threshold functions*. The problem of finding an LPB representation of a Boolean function if possible is called *threshold recognition problem* or *threshold synthesis problem*. The problem has an $O(m^7 t^5)$ algorithm using linear programming, where m is the dimension and t the number of clauses in the DNF (disjunctive normal form) input. There is also an entirely combinatorial procedure, which works by decomposing the DNF and "counting" the variable occurrences in it. We have implemented both algorithms and report here on the experiments. We were able to solve problems of up to 23 variables.

Note: This work was previously accepted as a *poster* but not as a full paper at the IWOCA workshop 2013. There will be a short summary in the IWOCA LNCS proceedings.

1. INTRODUCTION

A *linear pseudo-Boolean constraint* (LPB) [8, 1, 2, 5, 10, 9, 11, 12] is an expression of the form $a_1\ell_1 + \ldots + a_m\ell_m \geq d$. Here each ℓ_i is a *literal* of the form x_i or $\bar{x}_i \equiv 1 - x_i$, i.e., x_i becomes 0 if x_i is false and 1 if x_i is true, and vice versa for $\bar{x}_i$. Moreover, $a_1, \ldots, a_m, d$ are natural numbers.

An LPB can be used to represent a Boolean[1] function; e.g. $2x_1 + \bar{x}_2 + x_3 \geq 2$ represents the same function as the propositional formula $x_1 \vee (\neg x_2 \wedge x_3)$. It has been observed that a function can be often represented more compactly as a set of LPBs than as a *conjunctive* or *disjunctive* normal form (CNF or DNF) [5, 10]. E.g. the LPB $2x_1 + \bar{x}_2 + x_3 + x_4 \geq 2$ corresponds to the DNF $x_1 \vee (\neg x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (x_3 \wedge x_4)$.

Functions that can be represented by a single LPB are called *threshold functions* [17]. They have been studied intensively in the 1960s, but even very recently they have attracted interest due to various applications in artificial intelligence [3], electronic design automation [5], game theory [4], and several others [13].

The problem of recognising threshold functions and finding the LPB for a threshold function given as DNF is called *threshold recognition* or *threshold synthesis problem*. The problem is known to have an $O(m^7 t^5)$ algorithm using linear programming, where m is the dimension and t the number of clauses in the DNF [8]. In this paper, we report on an implementation of this solution to the problem.

The very comprehensive reference on Boolean functions [8] formulates the following research challenge:

> Is it possible to recognize threshold functions through an entirely combinatorial procedure, i.e., without resorting to the solution of [the linear program] [...]?

The authors explicitly mention that various researchers have worked on this problem and also cite our work [18]. Amazingly, none of these researchers was aware of the solution proposed by Coates *et al.* [7, 6], which we reinvented [18]. It works by decomposing the DNF and "counting" its variable occurrences. More precisely, the solution proposed by [7] consists of a basic procedure which finds the solution in many cases, plus what might be called a "repair" procedure coming into play if the basic procedure fails.

Concerning the basic procedure, ours [18] very much resembles that of [7], but is superior in that symmetries in the input formula are exploited, not only at the level of the entire formula as has been proposed in [6], but also at the level of subformulas that arise during the computation.

Concerning the "repair" procedure, Coates and Lewis [7] devote 23 pages to describing it and claim that it renders the overall procedure complete. Given that the art of writing pseudocode was less advanced 50 years ago, implementing the procedure is anything but straightforward and constitutes one of the contributions of this article, apart from the implementation of the basis procedure and the experiments. We have implemented a repair procedure inspired by the description of [7]. It achieves completeness up to $m = 7$, a recognition rate of at least 99% up to $m = 14$, and at least 80% up to $m = 23$. How the procedure might be rendered complete will be discussed later.

We have run experiments for up to $m = 23$. The combinatorial algorithm appears to scale considerably better than the LP algorithm, have a complexity of $O(m \cdot t)$, and can solve the biggest problems in a couple of seconds on average.

This paper is organised as follows. We continue with some preliminaries. Sec. 3 describes the linear programming algorithm, Sec. 4 the combinatorial procedure, Sec. 5 the experiments, and Sec. 6 concludes.

[1] Whenever we say "function" we mean "Boolean function".

2. PRELIMINARIES

We assume the reader to be familiar with the basic notions of propositional logic. We follow [5]. An **m-dimensional Boolean function** f is a function $Bool^m \rightarrow Bool$. A **linear pseudo-Boolean constraint** (LPB) is an inequality of the form

$$a_1\ell_1 + \ldots + a_m\ell_m \geq d \qquad a_i \in \mathbb{N}, d \in \mathbb{Z}, \ell_i \in \{x_i, \bar{x}_i\} \quad (1)$$

where $\bar{x}_i \equiv 1 - x_i$. We identify 0 with *false* and 1 with *true*. We call the a_i **coefficients** and d the **threshold**.

A **DNF** is a propositional formula of the form $c_1 \vee \ldots \vee c_n$ where each **clause** c_j is a conjunction of literals. Formally, a DNF is a set of sets of literals, i.e., the order of clauses and the order of literals within a clause are insignificant. For DNFs, we assume without loss of generality that no clause is a subset of another clause. We call a DNF *prime irredundant* if every clause is a prime implicant, i.e., if for clause c_1 there is no clause $c_2 \neq c_1$ such that $c_1 \vee c_2 = c_2$.

It is easy to see that an LPB can only represent *monotone* functions, i.e., functions represented by a DNF where each variable occurs in only one polarity. Hence any DNF containing a variable in different polarities is uninteresting for us. Without loss of generality, we assume that this polarity is positive.

Variables x and y are **symmetric** in ϕ if ϕ is equivalent to the formula obtained by exchanging x and y. A set of variables Y is **symmetric** in ϕ if each pair in Y is symmetric in ϕ.

3. THE LINEAR PROGRAMMING ALGORITHM

We shortly summarise the solution via linear programming [15, 8].

For some DNFs, it is possible to establish a complete order $\succeq$ on the variables which, intuitively speaking, has the following meaning: $x_i \succeq x_j$ iff starting from any given input tuple $X^* \in Bool^m$, setting x_i^* to true is more likely to make the DNF true than setting x_j^* true. The functions represented by such a DNF are called *regular*. The order is based on *occurrence patterns* [18] (the *Winder matrix* [8, 20]) and can be computed in time linear in $|\phi|$. We omit the actual definition of $\succeq$ here. We write $x_i \simeq x_k$ if $x_i \succeq x_k$ and $x_k \succeq x_i$.

Note that if $\sum_{i=1}^{m} a_i x_i \geq d$ is an LPB representing the DNF ϕ and $a_i = a_k$, then x_i, x_k are symmetric in ϕ; but $a_i \neq a_k$ does not imply that x_i, x_k are not symmetric. For example, $x_1 \vee x_2$ can be represented by $2x_1 + x_2 \geq 1$ or $x_1 + x_2 \geq 1$.

The algorithm first tests the input DNF for the regularity property. The property is weaker than the threshold property, and so if a DNF is not regular, then it is not convertible and we must give up.

If the DNF is regular, we use the *minimal true points* of the DNF, i.e. the true tuples where we cannot set any 1-value to 0 without making the point false. We also use the *maximal false points* defined analogously. Note that these together characterise the DNF uniquely. In general, no polynomial algorithm is known to find these points (which is no surprise since the general task is NP-complete [15]), but for the special case that the input DNF is prime irredundant, a polynomial algorithm for finding these points exists, because the true points can be read directly from the clauses. Then there exists a polynomial time procedure to find the maximal false points [8].

Then we can formulate the following linear program where the minimal true points are $x^1, \ldots, x^k$ and the maximal false points are $y^1, \ldots, y^l$:

$$\begin{aligned} \sum_{i=1}^{m} a_i x_i^j &\geq d \quad (1 \leq j \leq k) \\ \sum_{i=1}^{m} a_i y_i^j &< d \quad (1 \leq j \leq l) \\ a_i &\geq 0 \quad (1 \leq i \leq m) \end{aligned}$$

Note that the weights a_i are the variables in the LP formulation and the threshold is d. Finally, the linear program is passed to an LP solver. The reason for the complexity blow-up $O(m^7 t^5)$ is mainly due to the linear programming. The other parts run in $O(m^2 t)$, so the whole procedure gains from future improvements of linear programming. It should be mentioned that for most inputs the well-known simplex method for solving linear programs runs in linear time.

4. THE COMBINATORIAL ALGORITHM

In this section we present a combinatorial algorithm for the threshold synthesis problem first presented by Coates *et al.* [7, 6] and rediscovered later by Smaus [18]. The algorithm is divided into a basic algorithm (Part I [7]) which is correct but incomplete, and a backtracking procedure to obtain completeness (Part II and III [7]). We only describe Part I in detail here, but our implementation covers all parts.

Consider again the order $\succeq$ on the variables: it must be respected by any LPB (if there is one!) representing ϕ, i.e., $x_i \succeq x_k$ implies $a_i \geq a_k$. Note that $x_i \succ x_k$ implies $a_i > a_k$ but $x_i \simeq x_k$ does not imply $a_i = a_k$ due to integrality constraints.

Suppose we want to find an LPB representing ϕ, and we have already established the order of the coefficients. Assume the numbering of the variables is such $x_1 \succeq \ldots \succeq x_m$. Consider now the maximal set $X = \{x_1, \ldots, x_l\}$ such that $x_1 \simeq \ldots \simeq x_l$. We partition ϕ according to how many variables from X each clause contains. We then remove the variables from X from each clause, which gives $l+1$ subproblems. Theorem 5 states under which conditions solutions to these subproblems can be combined to an LPB for ϕ.

DEFINITION 1. *Let ϕ be a DNF and X a subset of its variables with $|X| = l$. If ϕ contains a clause $c \subseteq X$, then let $k_{\max}$ be the length of the longest such clause; otherwise let $k_{\max} := \infty$. For $0 \leq k \leq l$, we define $S(\phi, X, k)$ as the disjunction of clauses from ϕ containing exactly* $\min\{k, k_{\max}\}$ *variables from X, with those variables removed.*

When constructing the $S(\phi, X, k)$ from ϕ, we say that we *split away* the variables in X from ϕ.

EXAMPLE 2. *Let $\phi \equiv (x_1) \vee (x_2) \vee (x_3 \wedge x_4)$ and $X = \{x_1, x_2\}$. We have $k_{\max} = 1$. Then $S(\phi, X, 0) = (x_3 \wedge x_4)$, $S(\phi, X, 1) = true$ (i.e., the disjunction of twice the empty conjunction), and $S(\phi, X, 2) = true$.*

We must solve the $l+1$ subproblems in such a way that the resulting LPBs agree in all coefficients, and that the threshold difference of neighbouring LPBs is always the same. Before giving the theorem, we give two examples for illustration.

EXAMPLE 3. *Consider* $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ *and* $X = \{x_1\}$. *Then* $S(\phi, X, 0) = x_2 \wedge x_3 \wedge x_4$, *represented by* $x_2 + x_3 + x_4 \geq 3$. *Moreover,* $S(\phi, X, 1) = x_2 \vee x_3 \vee x_4$, *represented by* $x_2 + x_3 + x_4 \geq 1$.

Since the coefficients of the two LPBs agree, it turns out that ϕ *can be represented by* $2x_1+x_2+x_3+x_4 \geq 3$. *The coefficient of* x_1 *is given by the difference of the two thresholds, i.e.,* $3 - 1$.

EXAMPLE 4. *Consider* $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ *and* $X = \{x_1, x_2\}$. *We have* $S(\phi, X, 0) = \mathit{false}$, *represented by* $x_3 + x_4 \geq 4$, $S(\phi, X, 1) = x_3 \wedge x_4$, *represented by* $x_3 + x_4 \geq 2$, *and* $S(\phi, X, 2) = \mathit{true}$, *represented by* $x_3 + x_4 \geq 0$. *The DNF* ϕ *is represented by* $2x_1+2x_2+x_3+x_4 \geq 4$. *The coefficient of* x_1, x_2 *is given by* $4 - 2 = 2 - 0 = 2$ *(the thresholds are "equidistant").*

THEOREM 5. *Let* ϕ *be a DNF in variables* $x_1, \ldots, x_m$ *and suppose* $X = \{x_1, \ldots, x_l\}$ *are symmetric variables that are maximal w.r.t.* $\succeq$ *in* ϕ. *Then* ϕ *is represented by an LPB* $\sum_{i=1}^{m} a_i x_i \geq d$, *where* $a_1 = \ldots = a_l$, *iff for all* $k \in [0..l]$, *the DNF* $S(\phi, X, k)$ *is represented by* $\sum_{i=l+1}^{m} a_i x_i \geq d - k \cdot a_1$.

Unfortunately, Thm. 5 does not immediately dictate an algorithm for computing an LPB for a DNF if possible. The remaining problem is that a DNF might be represented by various LPBs, and so even if the LPBs computed recursively do not have agreeing coefficients and equidistant thresholds, one might find alternative LPBs (such as the non-obvious LPB for *false* in Ex. 4) so that Thm. 5 can be applied.

We now generalise LPBs by recording to what extent thresholds can be shifted without changing the meaning. To formulate this, we temporarily lift the restriction that coefficients and thresholds must be integers.

DEFINITION 6. *Given an LPB* $I \equiv \sum_{i=1}^{m} a_i x_i \geq d$, *we call* s *the* ***minimum threshold*** *of* I *if* s *is the smallest number (possibly* $-\infty$*) such that for any* $s' \in (s, d]$, *the LPB* $\sum_{i=1}^{m} a_i x_i \geq s'$ *represents the same function as* I. *We call* b *the* ***maximum threshold*** *if* b *is the biggest number (possibly* ∞*) such that* $\sum_{i=1}^{m} a_i x_i \geq b$ *represents the same function as* I. *We call* $b - s$ *the* ***gap*** *of* I.

Note that the minimum threshold of I is not a possible threshold of I. Since the minimum and maximum thresholds of an LPB are more informative than its actual threshold, we use the notation $\sum_{i=1}^{m} a_i x_i \geq (s, b]$ for denoting an LPB with minimum threshold s and maximum threshold b.

Theorem 5 suggests a recursive algorithm where, at least conceptually, in the base case we have at most 2^m trivial problems of determining an LPB.

EXAMPLE 7. *Consider* $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5)$. *To find an LPB for* ϕ, *we must find LPBs for* $S(\phi, \{x_1\}, 0)$ *and* $S(\phi, \{x_1\}, 1)$. *To find an LPB for* $S(\phi, \{x_1\}, 0)$, *we must find LPBs for* $S(S(\phi, \{x_1\}, 0), \{x_2\}, 0)$ *and* $S(S(\phi, \{x_1\}, 0), \{x_2\}, 1)$, *and so forth. Table 1 gives all the formulae for which we must find LPBs. For a concise notation we use some abbreviations which we explain using* $S(\cdot, x_{3..5}, 0) \equiv f$ *in the top-right corner: it stands for* $S((x_3 \wedge x_4 \wedge x_5), \{x_3, x_4, x_5\}, 0) \equiv \mathit{false}$, *i.e., the '$\cdot$' stands for the nearest* non-shaded *formula to the left, here* $(x_3 \wedge x_4 \wedge x_5)$. *Note how we arranged the subproblem formulae in the table: e.g.* $(x_3 \wedge x_4 \wedge x_5)$ *has* three *symmetric variables that are split away to obtain the subproblems to be solved, so these subproblems are located* three *columns to the right of* $(x_3 \wedge x_4 \wedge x_5)$. *The two shaded boxes in between contain the subproblems obtained by splitting away only* $\{x_3\}$, $\{x_3, x_4\}$, *resp.*

The algorithm we propose is not a purely recursive one, since the subproblems at each level must be solved in parallel. Explained using the example, we first find LPBs for the formulae in the rightmost column, which have 0 variables and hence we must determine 0 coefficients. Next to the left, we have formulae that contain (at most) x_5, and we determine LPBs representing these, where we use the same a_5 for all formulae! Then we determine a_4, and so forth.

Taking f.i. $(x_3 \wedge x_4 \wedge x_5)$ in Table 1, Thm. 5 suggests that a_3, a_4, a_5 should be equal (x_3, x_4, x_5 are symmetric) and determined in one go. However, since x_3, x_4, x_5 are not globally symmetric, one cannot determine a_3, a_4, a_5 in one go, but rather first a_5, then a_4, then a_3. Therefore, it is necessary to consider the formulae obtained by splitting away just x_3 and then x_4. These are the shaded formulae.

We call the formulae in column $l + 1$ the *l-successors*. Shaded formulae are called *auxiliary*, the others are called *main*. Formulae that have no further formulae to the right are called *final*.

DEFINITION 8. *Let* ϕ *be a DNF in* m *variables. Then* ϕ *is the* 0-***successor*** *of* ϕ. *Furthermore,* ϕ *is a* ***main*** *successor of* ϕ. *Moreover, if* ϕ' *is a main* n*-successor of* ϕ, *and* l *is maximal so that* $x_{n+1}, \ldots, x_{n+l}$ *are symmetric in* ϕ', *then for all* l', k *with* $1 \leq l' \leq l$ *and* $0 \leq k \leq l'$, *we say that* $S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k)$ *is an* $(n+l')$*-successor of* ϕ. *The* $(n+l)$*-successors are called* ***main****, and for* $l' < l$, *the* $(n+l')$*-successors are called* ***auxiliary****. A node that is a main node and* true *or* false *is called* final.

Note in particular $x_3 \vee x_4$ in column 3 in Table 1. It does not contain x_5, and so we obtain final 4-successors in the last-*but-one* column. Clearly, a final successor of ϕ is either *true* or *false*.

Generally, each non-final successor is associated with two formulae in the column right next to it, one shifted up and one down, obtained by splitting away the variable with the smallest index. In fact, Table 1 resembles a binary tree with the root displayed to the left. However, it is *not quite* a tree: take the node $x_4 \wedge x_5$ and the node f above it, for instance; the "upper child" of $x_4 \wedge x_5$ and the "lower child" of f coincide; they are both the f-node marked with †.

The following proposition explains this sharing.

PROPOSITION 9. *Assume* ϕ, ϕ', n, l *as in Def. 8. For* $0 < l' < l$ *and* $0 \leq k \leq l'$, *we have*

$$S(S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k), \{x_{n+l'+1}\}, 0) \equiv S(\phi', \{x_{n+1}, \ldots, x_{n+l'+1}\}, k)$$
$$S(S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k), \{x_{n+l'+1}\}, 1) \equiv S(\phi', \{x_{n+1}, \ldots, x_{n+l'+1}\}, k+1)$$

Taking $\phi' \equiv x_3 \wedge x_4 \wedge x_5$, e.g., the proposition says that $S(S(\phi', \{x_3\}, 0), \{x_4\}, 1)$ and $S(S(\phi', \{x_3\}, 1), \{x_4\}, 0)$ are both equal to $S(\phi', \{x_3, x_4\}, 1) \equiv \mathit{false}$.

If we took a naïve approach where we always split away one variable at a time, we would have a table (tree) with 32 ($= 2^m$) formulae in the rightmost column. Thanks to the sharing, we only have 12 final formulae instead.

ϕ	$S(\cdot, x_1, 0)$ $\equiv (x_2 \wedge x_3) \vee$ $(x_2 \wedge x_4) \vee$ $(x_3 \wedge x_4 \wedge x_5)$	$S(\cdot, x_2, 0) \equiv$ $(x_3 \wedge x_4 \wedge x_5)$	$S(\cdot, x_3, 0) \equiv f$ $S(\cdot, x_3, 1)$ $\equiv (x_4 \wedge x_5)$	$S(\cdot, x_{3..4}, 0) \equiv f$ $S(\cdot, x_{3..4}, 1) \equiv f$ † $S(\cdot, x_{3..4}, 2) \equiv x_5$	$S(\cdot, x_{3..5}, 0) \equiv f$ $S(\cdot, x_{3..5}, 1) \equiv f$ $S(\cdot, x_{3..5}, 2) \equiv f$ $S(\cdot, x_{3..5}, 3) \equiv t$
		$S(\cdot, x_2, 1) \equiv$ $x_3 \vee x_4$	$S(\cdot, x_3, 0) \equiv x_4$ $S(\cdot, x_3, 1) \equiv t$	$S(\cdot, x_{3..4}, 0) \equiv f$ $S(\cdot, x_{3..4}, 1) \equiv t$ $S(\cdot, x_{3..4}, 2) \equiv t$	
	$S(\cdot, x_1, 1)$ $\equiv x_2 \vee x_3$ $\vee x_4 \vee x_5$	$S(\cdot, x_2, 0) \equiv$ $x_3 \vee x_4 \vee x_5$ $S(\cdot, x_2, 1) \equiv t$	$S(\cdot, x_{2..3}, 0) \equiv$ $x_4 \vee x_5$ $S(\cdot, x_{2..3}, 1) \equiv t$ $S(\cdot, x_{2..3}, 2) \equiv t$	$S(\cdot, x_{2..4}, 0) \equiv x_5$ $S(\cdot, x_{2..4}, 1) \equiv t$ $S(\cdot, x_{2..4}, 2) \equiv t$ $S(\cdot, x_{2..4}, 3) \equiv t$	$S(\cdot, x_{2..5}, 0) \equiv f$ $S(\cdot, x_{2..5}, 1) \equiv t$ $S(\cdot, x_{2..5}, 2) \equiv t$ $S(\cdot, x_{2..5}, 3) \equiv t$ $S(\cdot, x_{2..5}, 4) \equiv t$

Table 1: The recursive problems of Ex. 7

The initial work by Coates and Lewis [7] is not quite as naïve as just described, in that nodes containing *true* or *false* are not expanded further (unlike in Table 1). But symmetries that allow for node sharing are not considered.

Coates et al. [6] have improved this method by considering symmetries, but only "global" symmetries that are present in the initial DNF ϕ, not "local" symmetries that appear only after splitting. Even for initial DNFs without any global symmetries, many local symmetries usually appear after splitting.

In our code, it is possible to switch on an option so that nodes containing *true* or *false* are not expanded further. Combining both ideas, symmetry and not expanding *true* or *false* any further, lead to considerable savings, as we discuss in Sec. 5.

The following theorem states if and how one can find the next coefficient and thresholds for representing all k-successors of ϕ provided one has coefficients and thresholds for representing all $(k+1)$-successors.

THEOREM 10. *Assume ϕ as in Thm. 5 and some k with $0 \leq k \leq m-1$, and let Φ_k be the set of k-successors of ϕ. For every non-final $\phi' \in \Phi_k$, suppose we have two LPBs $\sum_{i=k+2}^{m} a_i x_i \geq (s_{\phi' 0}, b_{\phi' 0}]$ and $\sum_{i=k+2}^{m} a_i x_i \geq (s_{\phi' 1}, b_{\phi' 1}]$, representing $S(\phi', \{x_{k+1}\}, 0)$ and $S(\phi', \{x_{k+1}\}, 1)$, resp.*

If it is possible to choose a_{k+1} such that

$$\max_{\phi' \in \Phi_k} (s_{\phi' 0} - b_{\phi' 1}) < a_{k+1} < \min_{\phi' \in \Phi_k} (b_{\phi' 0} - s_{\phi' 1}), \qquad (3)$$

then for all $\phi' \in \Phi_k$, the LPB $\sum_{i=k+1}^{m} a_i x_i \geq (s_{\phi'}, b_{\phi'}]$ represents ϕ', where

$$\begin{array}{ll} s_{\phi'} = \max\{s_{\phi' 0}, s_{\phi' 1} + a_{k+1}\}, & \\ b_{\phi'} = \min\{b_{\phi' 0}, b_{\phi' 1} + a_{k+1}\} & \text{for non-final } \phi' \end{array} \qquad (4)$$

$$\begin{array}{ll} s_{\phi'} = -\infty,\ b_{\phi'} = 0 & \text{for } \phi' \equiv \textit{true} \\ s_{\phi'} = \sum_{i=k+1}^{m} a_i,\ b_{\phi'} = \infty & \text{for } \phi' \equiv \textit{false} \end{array} \qquad (5)$$

If $\max_{\phi' \in \Phi_k} (s_{\phi' 0} - b_{\phi' 1}) \geq \min_{\phi' \in \Phi_k} (b_{\phi' 0} - s_{\phi' 1})$, then no a_{k+1}, $s_{\phi'}$, $b_{\phi'}$ exist such that $\sum_{i=k+1}^{m} a_i x_i \geq (s_{\phi'}, b_{\phi'}]$ represents ϕ' for all $\phi' \in \Phi_k$.

Following [7], we call the interval for a_{k+1} given by (3) the **range** of a_{k+1}.

The m-successors of ϕ are represented by LPBs with an empty sum as l.h.s.: $\sum_{i=m+1}^{m} a_i x_i \geq (0, \infty]$ for *false* and $\sum_{i=m+1}^{m} a_i x_i \geq (-\infty, 0]$ for *true*. Then we proceed using Thm. 10, in each step choosing an arbitrary a_{k+1} fulfilling (3).

EXAMPLE 11. *Consider again Ex. 7. Table 2 is arranged in strict correspondence to Table 1 and shows LPBs for all successors of Φ. In the top line we give the l.h.s. of the LPBs, which is of course the same for each LPB in a column. In the main table, we list the minimum and maximum threshold of each formula.*

In the first step, applying (3), we have to choose a_5 so that

$$\begin{aligned} &\max\{0-\infty, 0-\infty, 0-0, \quad 0-0, -\infty-0, -\infty-0, \\ &\quad -\infty-0\} < a_5 < \min\{\infty-0, \infty-0, \infty--\infty, \\ &\quad \infty--\infty, 0--\infty, 0--\infty, 0--\infty\}. \end{aligned}$$

Choosing $a_5 = 1$ will do. The minimum and maximum thresholds in column 5 are computed using (4); e.g. the topmost $(1, \infty]$ is $(\max\{0, 0+1\}, \min\{\infty, \infty+1\}]$.

In the next step, we have to choose a_4 so that

$$\begin{aligned} &\max\{1-\infty, 1-1, \quad 1-0, -\infty-0, \quad 0-0, -\infty-0, \\ &\quad -\infty-0\} < a_4 < \min\{\infty-1, \infty-0, \\ &\quad \infty--\infty, 0--\infty, \quad 1--\infty, 0--\infty, 0--\infty\}. \end{aligned}$$

Choosing $a_4 = 2$ will do. Note that the bound $1-0 < a_4$ comes from the middle box of the fifth column and thus ultimately from $x_3 \vee x_4$. Our algorithm enforces that $a_4 > a_5$, which must hold for an LPB representing $x_3 \vee x_4$.

In the next step, a_3 can also be chosen to be any number > 1 so we choose 2 again. In the next step, $2 < a_2 < 4$ must hold so we choose $a_2 = 3$. Finally, $3 < a_1 < 5$ must hold so we choose $a_1 = 4$. We obtain the LPB $4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \geq (4, 5]$.

We have seen in the example how our algorithm works. However, since the choice of a_{k+1} is not unique in general, a bad choice of a_{k+1} might later lead to non-applicability of Thm. 10, making the algorithm incomplete. We will discuss this below.

Another problem seems to be that a_{k+1} could be forced to be between neighbouring integers, in which case it cannot be an integer itself. However, in this case, one can multiply all LPBs of the current system by 2 before proceeding so that a_{k+1} can be chosen to be an integer.

From the construction of the successors it follows that all formulae in a column together have size less than all

$4x_1 + 3x_2 +$ $2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$3x_2 +$ $2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$2x_3 + 2x_4 +$ $x_5 \geq \ldots$	$2x_4 +$ $x_5 \geq \ldots$	$x_5 \geq \ldots$	$\sum_{i=6}^{5} a_i x_i$ $\geq \ldots$
				$(1, \infty]$	$(0, \infty]$
			$(3, \infty]$	$(1, \infty]$	$(0, \infty]$
		$(4, 5]$	$(2, 3]$	$(0, 1]$	$(0, \infty]$
	$(4, 5]$				$(-\infty, 0]$
			$(1, 2]$	$(1, \infty]$	
		$(1, 2]$	$(-\infty, 0]$	$(-\infty, 0]$	
$(4, 5]$				$(-\infty, 0]$	
				$(0, 1]$	$(0, \infty]$
			$(0, 1]$	$(-\infty, 0]$	$(-\infty, 0]$
	$(0, 1]$	$(0, 1]$	$(-\infty, 0]$	$(-\infty, 0]$	$(-\infty, 0]$
		$(-\infty, 0]$	$(-\infty, 0]$	$(-\infty, 0]$	$(-\infty, 0]$
					$(-\infty, 0]$

Table 2: LPBs for Ex. 7

formulae in the column to the left of it, so that the entire table has size less than $|\phi| \cdot (m+1)$. One can thus show that the complexity of the algorithm is polynomial in the size of ϕ, while the size of ϕ itself can be exponential in m.

4.1 The Symmetry Issue

The symmetry of variables is very important in this section, in Theorem 5, Def. 8, and implicitly also in Theorem 10. Whenever we have variables with the same occurrence pattern, then these variables must be symmetric if the DNF represents a threshold function. Put differently, our way of exploiting symmetries by node sharing is incorrect if the variables in question are not actually symmetric. The symmetry can be checked while constructing the "tree": whenever the lower child of one node ought to be identical to the upper child of another node, as described above, then one must generate both children and check that they are indeed identical. If not, the DNF is not a threshold function.

In our code, this check is optional, which is good for experimentation. We observe, and this was not obvious to us, that the test is really necessary, i.e., there are examples where without the test, we wrongly obtain a result.

A simple such example is the following: $(x_1 \wedge x_2) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4)$. The trick for designing this example is to start with the DNF for $x_1 + x_2 + x_3 + x_4 \geq 2$ and remove the two clauses $(x_1 \wedge x_3)$ and $(x_2 \wedge x_4)$. By doing so, the occurrence pattern for each variable is $\{\!\![2, 2]\!\!\}$; x_1 and x_3 are symmetric to each other; x_2 and x_4 are symmetric to each other; but x_1 or x_3 are not symmetric to x_2 or x_4.

Without the symmetry test, the program wrongly computes the result $x_1 + x_2 + x_3 + x_4 \geq 2$.

Note that the example crucially depends on the variable numbering, i.e., the order in which the variables are treated in the "tree" construction (which is arbitrary a-priori since the variables all have the same occurrence patterns). For a different ordering, even without the symmetry test, no result will be computed, because some numerical constraint will not be satisfiable.

The test has a cost, of course (switching it off may make the code run around 20% faster), but nevertheless, exploiting the symmetries helps limit the combinatorial "explosion" in constructing the "tree". As future work, we want to make the test more efficient by using a sorted representation of clauses.

4.2 Completion According to [7]

Concerning completeness, Coates and Lewis propose a "repair" procedure which they take 23 pages to describe! We briefly explain the main idea of this procedure, using Table 2 for illustration: consider the third column. There are two pairs of sibling nodes in this column: $(4, 5]$, $(1, 2]$, and $(0, 1]$, $(-\infty, 0]$. According to the basic procedure described above, each pair enforces a lower and upper bound on the choice of a_2: $4 - 2 < a_2 < 5 - 1$ and $0 - 0 < a_2 < 1 - -\infty$, respectively.

The generalisation lies in recognising that not just siblings, but any pair of nodes in the column enforces bounds on a certain coefficient combination. E.g., the pair $(1, 2]$, $(-\infty, 0]$ enforces $1 - 0 < a_1 < 2 - -\infty$. The repair procedure looks at some nodes in columns already treated by the basic procedure to see if those nodes imply contradicting bounds for some coefficient combination. E.g. it might be that one can read off some column that $4 < a_1 - a_3 + a_4 < 3$ which is a contradiction showing that some previous choice of coefficient was bad.

We do not go into any details, but we want to make some remarks:

1. In spite of trying hard, we do not understand the description of [7] sufficiently well to be able to state precisely in how far it corresponds to our implementation.

2. Our current implementation is still incomplete.

3. Taken to the extreme, the solution of [7] amounts to collecting at least the constraints of the linear program (see Sec. 3), showing that there is a kind of continuum between the "numerically flavoured" LP algorithm and the "combinatorial" algorithm of this section. This suggests that one might design a combination of the two algorithms.

4. In [7] it is conjectured that the basic procedure is complete up to about $m = 9$; our experiments reveal that it is already incomplete for $m = 6$.

5. We had more than a dozen rounds of extending our code to cover some extremely rare cases — an exercise in combinatorics indeed.

6. During all those extensions one charming property of our repair procedure was preserved: the procedure only looks at a constant number of nodes per column (in fact, at most six)! Readily sacrificing this property might be the conceptually easiest, but computationally complex, way to achieve completeness.

7. It can happen that a run of the combinatorial algorithm requires several calls to the "repair" procedure.

5. EXPERIMENTS

Both algorithms have been implemented in C++ based on a previous implementation in Java [16, 19].

In order to evaluate algorithms for the threshold recognition problem, one needs benchmark DNFs. In [14], all monotone (unate) Boolean functions of up to 5 variables are considered as input. For 6 variables, 200,000 "random" monotone functions are chosen as input. However, we believe that for testing purposes, it is better to use DNFs for which it is *known* that they are threshold functions. To do so, one should generate LPBs and convert them into DNFs.

We tested both algorithms by generating LPBs where the coefficients are all positive and non-increasing from a_1 to a_m. Up to $m = 7$, we generated all LPBs up to equivalence (28262 LPBs for $m = 7$). For $m = 8$, we enumerated 248k LPBs which is a bit less than 10% of the existing LPBs.

For bigger m, the number of LPBs becomes too big and so we generated a "random" sample of LPBs. We do not describe our method in detail here, but it was guided by the following principles: (1) the method should be simple and avoid ad-hoc choices; (2) it should be possible to calibrate how big the increase from the smallest to the biggest coefficient is; (3) every coefficient vector (for given m) should have a positive probability, however small, of being generated.

For $m = 9, 10$, we generated 30000 benchmarks, for $m =$ 11 to 15, we generated 3000, for $m = 16$ to 20, we generated 300, and for $m = 21$ to 22, we generated 60. To give some idea, we blindly picked three of the generated coefficient sequences (which were then transformed to DNFs to create the benchmarks) for $m = 15$: 3 3 3 2 2 2 1 1 1 1 1 1 1 1 1, 135 127 55 43 36 32 31 27 25 14 13 11 6 3 1, 8 7 7 7 6 6 5 5 4 4 4 2 2 2 2.

Figure 1 shows the failure rate of the current implementation of the combinatorial algorithm, i.e., which percentage of the benchmarks it was not able to solve due to its incompleteness. The x-axis shows m, and the y-axis the percentage in logarithmic scale. Note that for $m \leq 7$, the failure rate is 0, although strictly speaking this cannot be displayed in logarithmic scale. Up to $m = 14$, the failure rate is less than 1% while rising up to 18.3% for $m = 22$. We are still working on the problem of making the repair procedure complete while preserving the property that its effort is constant per column.

Figure 2 shows the percentage of problems that required at least one call to the repair procedure to be solved. The x-axis shows m, and the y-axis the percentage in linear scale. The picture is not so clear, but it appears that for higher m the repair procedure is required more often. As future work, we want to look at heuristics that guess the right coefficient in the first place, thus reducing the need for any repair procedure.

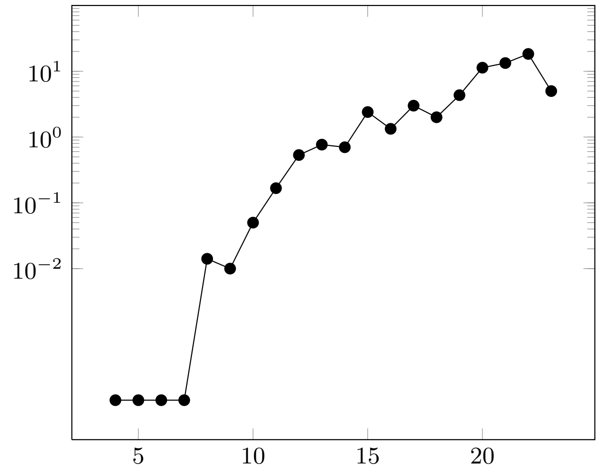

Figure 1: Failure rate in %

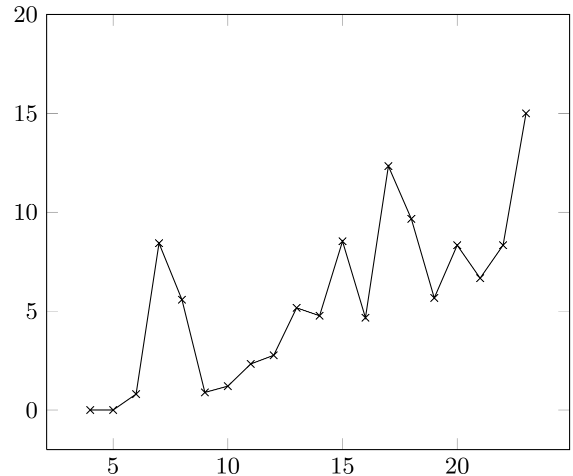

Figure 2: Percentage of the successful calls requiring repair

Figure 3 shows the runtime per problem for both algorithms in ms, as well as the problem size. We used a machine with an Intel Core i7-2620M processor running at 2.7 GHz with 3 GB RAM. As before, the x-axis shows m. The y-axis is in logarithmic scale. We first observe that the combinatorial algorithm could solve problems up to $m = 23$ in a couple of seconds, while the LP algorithm appears to scale worse and needs around 30 seconds for the biggest problems.[2] Second, the runtime seems to be exponential in m. Let us now discuss the problem size. Note that the input to our procedure is a DNF. The combinatorics wants it that the size of the (randomly generated) DNFs grows exponentially in m. More precisely, we define the size of a DNF as the number of clauses in it (denoted t in Sec. 3) multiplied by the average number of literals per clause, or equivalently, simply the number of literal occurrences in the DNF. The average number of clauses appears to be roughly $0.46 \cdot 1.633^m$, while

[2]The LP algorithm ran out of memory for $m = 23$.

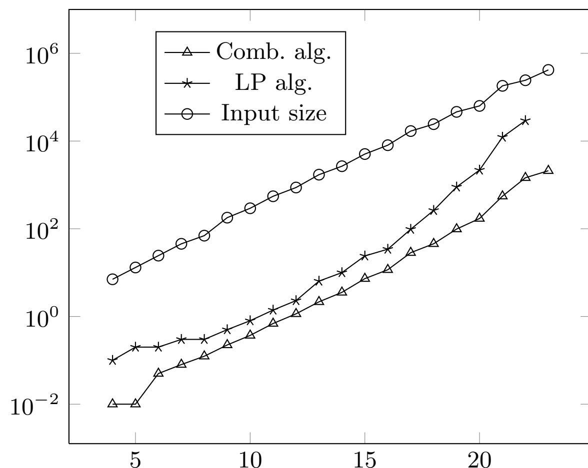

Figure 3: Runtime

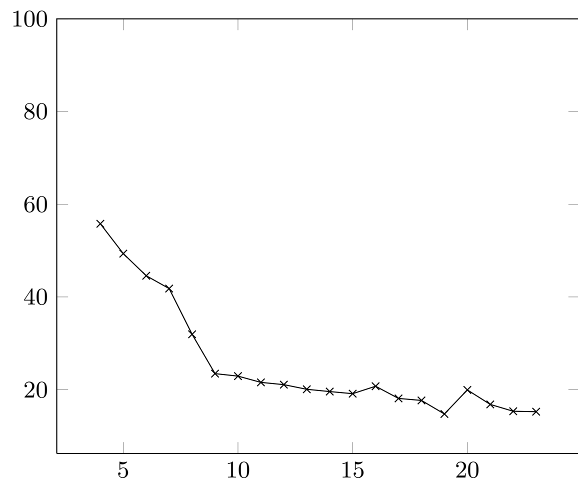

Figure 4: Number of final true nodes / number of clauses

the average clause length is $\frac{m}{2}$ with remarkable precision, so that the input size is $O(m \cdot t)$. The size, around 243000 for $m = 22$, is shown in the figure. The fact that the curve is almost a perfect straight line and appears to be parallel to the curve for the runtime of the combinatorial algorithm shows that the input size increases at the same rate as that runtime, which means that the algorithm appears to run in time linear to the input, whereas the LP algorithm performs worse.

We now discuss the savings due to symmetry, as mentioned in Sec. 4. A simple and appropriate way of measuring these saving is to look at the final *true* nodes in the "tree". In the original approach by Coates and Lewis [7], nodes containing *true* and *false* are not expanded any further, but symmetries are not exploited. With that approach, one can see that every final *true* node corresponds to exactly one clause in the original DNF, since the *true* node is obtained by successively splitting away all the variables from the clause.

Now, in our approach, we also do not expand *true* and *false* any further, but we also exploit symmetries. Thus if we take the number of final *true* nodes in our "tree" and divide it by the number of clauses in the DNF, then this is a good measure for how big our "tree" is relative to the size it would have without exploiting the symmetry. Figure 4 shows this relative size (given as percentage) for our benchmarks. It is not completely clear but it might be that in theory this relative size goes asymptotically to a value around 15%.

6. CONCLUSION

In this paper, we have presented the implementations of two algorithms for the threshold recognition problem. The first algorithm works by a translation to linear programming, while the second is combinatorial, i.e., it works by taking the formula apart and counting the variables in it in a certain way. The second algorithm has first been presented in [7, 6] and rediscovered later [18], although with a more powerful treatment of the symmetries in a formula.

It is unsatisfactory that our current implementation of the combinatorial algorithm is incomplete. However, recently a method has been presented [14] which appears to be "much more" incomplete. For 6 variables, it identifies only 70% of the threshold functions. In experiments, the authors were able to identify 36 10-variable threshold functions among 10000 candidate functions — unfortunately, nobody knows how many of those 10000 functions actually were threshold functions, so that the statement says little about the real power of their method. As stated in 1961 [7], problems of this size can still be solved by hand: 30 minutes for a 6-variable function, 4 hours for a 12-variable function. The work uses the *Chow parameters* [8] of each variable as basis for deciding the coefficient. We plan to investigate similar ideas, but only to provide our algorithm with some *heuristic value* for each coefficient, hoping that this will reduce the need for backtracking.

7. REFERENCES

[1] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 450–457. ACM, 2002.

[2] Peter Barth. Linear 0-1 inequalities and extended clauses. In Andrei Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 40–51. Springer-Verlag, 1993.

[3] Peter Barth and Alexander Bockmayr. Solving 0-1 problems in CLP(PB). In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*. IEEE, 1993.

[4] Stefan Bolus. Power indices of simple games and vector-weighted majority games by means of binary decision diagrams. *European Journal of Operational Research*, 210(2):258–272, 2011.

[5] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835. ACM, 2003.

[6] Clarence L. Coates, R. B. Kirchner, and Philip M. Lewis II. A simplified procedure for the realization of

linearly-separable switching functions. *IRE Transactions on Electronic Computers*, 1962.

[7] Clarence L. Coates and Philip M. Lewis II. Linearly-separable switching functions. *Journal of Franklin Institute*, 272:366–410, 1961. Also in an expanded version, GE Research Laboratory, Schenectady, N.Y., Technical Report No.61-RL-2764E.

[8] Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, May 2011.

[9] Heidi E. Dixon. *Automating Pseudo-Boolean Inference within a DPLL Framework.* PhD thesis, University of Oregon, 2004.

[10] Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review*, 15:31–45, 2000.

[11] Martin Fränzle and Christian Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In Moshe Y. Vardi and Andrei Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNCS*, pages 302–316. Springer-Verlag, 2003.

[12] Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007.

[13] Tejaswi Gowda and Sarma B. K. Vrudhula. Decomposition based approach for synthesis of multi-level threshold logic circuits. In *ASP-DAC*, pages 125–130. IEEE, 2008.

[14] Ashok Kumar Palaniswamy, Manoj Kumar Goparaju, and Spyros Tragoudas. Scalable identification of threshold logic functions. In R. Iris Bahar, Fabrizio Lombardi, David Atienza, and Erik Brunvand, editors, *ACM Great Lakes Symposium on VLSI*, pages 269–274. ACM, 2010.

[15] Uri N. Peled and Bruno Simeone. Polynomial-time algorithms for regular set-covering and threshold synthesis. In *Discrete Applied Mathematics*, volume 12, pages 57–69, 1985.

[16] Christian Schilling. Solving the Threshold Synthesis Problem of Boolean Functions by Translation to Linear Programming. Bachelor thesis, Universität Freiburg, 2011.

[17] Ching Lai Sheng. *Threshold Logic.* Academic Press, 1969.

[18] Jan-Georg Smaus. On Boolean functions encodable as a single linear pseudo-Boolean constraint. In Pascal Van Hentenryck and Laurence Wolsey, editors, *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *LNCS*, pages 288–302. Springer-Verlag, 2007.

[19] Fabian Wenzelmann. Solving the Threshold Synthesis Problem of Boolean Functions by a Combinatorial Algorithm. Bachelor thesis, Universität Freiburg, 2011.

[20] Robert O. Winder. *Threshold Logic.* PhD thesis, Department of Mathematics, Princeton University, Princeton, U.S.A., 1962.