JAR manuscript No.
(will be inserted by the editor)

A Framework for Certified Boolean Branch-and-Bound Optimization

Javier Larrosa · Robert Nieuwenhuis · Albert Oliveras · Enric Rodríguez-Carbonell

Received: date / Accepted: date

Abstract We consider optimization problems of the form $(S, cost)$, where S is a clause set over Boolean variables $x_1 \dots x_n$, with an arbitrary cost function $cost\colon \mathbb{B}^n \to \mathbb{R}$, and the aim is to find a model A of S such that $cost(A)$ is minimized.

Here we study the generation of *proofs of optimality* in the context of branch-and-bound procedures for such problems. For this purpose we introduce $\texttt{DPLL}_{\texttt{BB}}$, an abstract DPLL-based branch-and-bound algorithm that can model optimization concepts such as cost-based propagation and cost-based backjumping.

Most, if not all, SAT-related optimization problems are in the scope of $\texttt{DPLL}_{\texttt{BB}}$. Since many of the existing approaches for solving these problems can be seen as instances, $\texttt{DPLL}_{\texttt{BB}}$ allows one to formally reason about them in a simple way and exploit the enhancements of $\texttt{DPLL}_{\texttt{BB}}$ given here, in particular its uniform method for generating independently verifiable optimality proofs.

Keywords SAT · optimization · proofs

1 Introduction

An important issue on algorithms for Boolean satisfiability is their ability to provide proofs of unsatisfiability, so that also negative answers can be verified with a trusted independent proof checker. Many current SAT solvers provide this feature typically by writing (with little overhead) a trace file from which a resolution proof can be reconstructed and checked.

In this paper we address a related topic. We take a very general class of Boolean optimization problems and consider the problem of computing the best model of a CNF with respect to a cost function and, additionally, a proof of its optimality. The purpose of the paper is to provide a general solving framework that is faithful to state-of-the-art branch-and-bound solvers and where it is simple to reason about them and to generate optimality proofs. We show how branch-and-bound algorithms can provide proofs with

Partially supported by Spanish Min. of Science and Innovation through the projects TIN2009-13591-C02-01 and TIN2007-68093-C02-01 (LogicTools-2).

From Technical Univ. of Catalonia, Barcelona

little overhead, as in the SAT case. To the best of our knowledge, no existing solvers offer this feature.

The first contribution of the paper is an abstract DPLL[1]-like branch-and-bound algorithm ($\texttt{DPLL}_{\texttt{BB}}$) that can deal with most, if not all, Boolean optimization problems considered in the literature. $\texttt{DPLL}_{\texttt{BB}}$ is based on standard abstract DPLL rules and includes features such as propagation, backjumping, learning or restarts. The essential difference between classical DPLL and its branch-and-bound counterpart is that the rules are extended from the usual SAT context to the optimization context by taking into account the cost function to obtain entailed information. Thus, $\texttt{DPLL}_{\texttt{BB}}$ can model concepts such as, e.g., cost-based propagation and cost-based backjumping. To exploit the cost function in the search with these techniques, $\texttt{DPLL}_{\texttt{BB}}$ assumes the existence of a lower bounding procedure that, additionally to returning a numerical lower bound, provides a reason for it, i.e., a (presumably short) clause whose violation is a sufficient condition for the computed lower bound, see [19, 21].

The second contribution of the paper is the connection between a $\texttt{DPLL}_{\texttt{BB}}$ execution and a proof of optimality. We show that each time that $\texttt{DPLL}_{\texttt{BB}}$ backjumps due to a soft conflict (i.e. the lower bound indicates that it is useless to extend the current assignment) we can infer a cost-based lemma, which is entailed from the problem. By recording these lemmas (among others), we can construct a very intuitive optimality proof.

This work could have been cast into the framework of SAT Modulo Theories (SMT) with a sequence of increasingly stronger theories [24]. There is already literature on generating proofs in SMT. For instance, in [7] the generation of unsatisfiable cores for SMT was analyzed; and in [23], a new formalism for encoding proofs allowing efficient proof checking was presented. However, the generation of proofs for SMT *with theory strengthening* has not been worked out so far, and would in any case obfuscate the simple concept of proof we have here. Also, we believe that in its current form, the way we have integrated the concepts of lower bounding and cost-based propagation and learning is far more useful and accessible to a much wider audience.

This paper is structured as follows. In Section 2 we give some basic notions and preliminary definitions. In Section 3 the $\texttt{DPLL}_{\texttt{BB}}$ procedure is presented, whereas in Section 4 we develop the framework for the generation of proof certificates. Section 5 shows several important instances of problems that can be handled with $\texttt{DPLL}_{\texttt{BB}}$. In Section 6 we report on the results of a prototypical implementation of the techniques presented here, which demonstrate the feasibility of the approach. In Section 7 other optimization-related problems are introduced, together with the corresponding extensions of the framework. Finally Section 8 gives conclusions of this work and points out directions for future research. Parts of this paper were presented in a short preliminary form (with less examples and without experiments and proofs) at the SAT'2009 conference [16].

2 Preliminaries

We consider a fixed set of Boolean variables $\{x_1, \ldots, x_n\}$. *Literals*, denoted by the (subscripted, primed) letter l are elements of the set $\{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$. The *negation of a literal* l, written $\neg l$, denotes $\neg x$ if l is a variable x, and x if l is $\neg x$. A *clause* (denoted by the letters $C, D, \ldots$) is a disjunction of literals $l_1 \vee \ldots \vee l_m$.

[1] For consistency with [25] we call it DPLL, although the presented abstract framework models CDCL SAT Solvers [22] rather than the original DPLL algorithm.

The *empty clause* (the disjunction of zero literals) will be noted $\square$. A (partial truth) *assignment* I is a set of literals such that $\{x, \neg x\} \subseteq I$ for no x. An assignment I is *total* if $\{x, \neg x\} \cap I \neq \emptyset$ for all x.

A literal l is *true* in I if $l \in I$, *false* in I if $\neg l \in I$, and *undefined* in I otherwise. A clause C is true in I if at least one of its literals is true in I, false in I if all its literals are false in I, and undefined in I otherwise. Note that the empty clause is false in every assignment I. We write $I \models l$ if the literal l is true in I and $I \models C$ if a clause C is true in I. Sometimes we will write $\neg I$ to denote the clause that is the disjunction of the negations of the literals in I. Similarly, we write $I \models \neg C$ to indicate that all literals of a clause C are false in I. A clause set S is true in I if all its clauses are true in I; if I is also total, then I is called a *total model* of S, and we write $I \models S$.

We consider the following class of problems, which covers a broad spectrum of instances (see Section 5):

Definition 1 A *Boolean optimization problem* is a pair (S, cost), where S is a clause set, *cost* is a function $\mathit{cost} \colon \mathbb{B}^n \to \mathbb{R}$, and the goal is to find a model A of S such that $\mathit{cost}(A)$ is minimized.

Definition 2 A *cost clause* is an expression of the form $C \vee c \geq k$ where C is a clause and $k \in \mathbb{R}$.

A cost clause $C \vee c \geq k$ may be better understood with its equivalent notation $\neg C \longrightarrow c \geq k$. Its intended meaning is that if C is falsified in a model A of the given clause set S, then $\mathit{cost}(A)$ is greater than or equal to k (so the lower case c is just a placeholder meaning something like "the cost of the model"). This is indeed the case if the cost clause is *entailed* by (S, cost):

Definition 3 Let (S, cost) be an optimization problem. A cost clause $C \vee c \geq k$ is *entailed by* (S, cost) if $\mathit{cost}(A) \geq k$ for every model A of S such that $A \models \neg C$.

Definition 4 Given an optimization problem (S, cost), a real number k is called a *lower bound* for an assignment I if $\mathit{cost}(A) \geq k$ for every model A of S such that $I \subseteq A$.

A *lower bounding procedure* lb is a procedure that, given an assignment I, returns a lower bound k, denoted $lb(I)$, and a cost clause of the form $C \vee c \geq k$, called the *lb-reason* of the lower bound, such that $C \vee c \geq k$ is entailed by (S, cost) and $I \models \neg C$.

Any procedure that can compute a lower bound k for a given I can be extended to a lower bounding procedure: it suffices to generate $\neg I \vee c \geq k$ as the lb-reason. But generating *short* lb-reasons is important for efficiency reasons, and in Section 5 we will see how this can be done for several classes of lower bounding methods.

3 Abstract Branch and Bound

3.1 $\texttt{DPLL}_{\texttt{BB}}$ Procedure

The $\texttt{DPLL}_{\texttt{BB}}$ procedure is modeled by a transition relation, defined by means of rules over states.

Definition 5 A $\texttt{DPLL}_{\texttt{BB}}$ *state* is a 4-tuple $I \parallel S \parallel k \parallel A$, where:
I is a sequence of literals representing the current partial assignment,
S is a finite set of classical clauses (i.e. not cost clauses),
$k \in \mathbb{R} \cup \{\infty\}$ is a number representing the best-so-far cost,
A is the best-so-far model of S (i.e. $cost(A) = k$).
Some literals l in I are annotated as *decision literals* and written l^{d}.

Note that the *cost* function and the variable set are not part of the states, since they do not change over time (they are fixed by the context).

Definition 6 The $\texttt{DPLL}_{\texttt{BB}}$ *system* consists of the following rules:

Decide :
$$I \parallel S \parallel k \parallel A \implies I\, l^{\mathsf{d}} \parallel S \parallel k \parallel A \quad \textbf{if} \quad \{\, l \text{ is undefined in } I$$

UnitPropagate :
$$I \parallel S \parallel k \parallel A \implies I\, l \parallel S \parallel k \parallel A \quad \textbf{if} \quad \begin{cases} C \vee l \in S,\ I \models \neg C \\ l \text{ is undefined in } I \end{cases}$$

Optimum :
$$I \parallel S \parallel k \parallel A \implies \mathit{OptimumFound} \quad \textbf{if} \quad \begin{cases} C \in S,\ I \models \neg C \\ \text{no decision literals in } I \end{cases}$$

Backjump :
$$I\, l^{\mathsf{d}}\, I' \parallel S \parallel k \parallel A \implies I\, l' \parallel S \parallel k \parallel A \quad \textbf{if} \quad \begin{cases} C \vee l' \in S,\ I \models \neg C \\ l' \text{ is undefined in } I \end{cases}$$

Learn :
$$I \parallel S \parallel k \parallel A \implies I \parallel S, C \parallel k \parallel A \quad \textbf{if} \quad \{\, (S, cost) \text{ entails } C \vee c \geq k$$

Forget :
$$I \parallel S, C \parallel k \parallel A \implies I \parallel S \parallel k \parallel A \quad \textbf{if} \quad \{\, (S, cost) \text{ entails } C \vee c \geq k$$

Restart :
$$I \parallel S \parallel k \parallel A \implies \emptyset \parallel S \parallel k \parallel A$$

Improve :
$$I \parallel S \parallel k \parallel A \implies I \parallel S \parallel k' \parallel I \quad \textbf{if} \quad \{\, I \models S \text{ and } cost(I) = k' < k$$

As we will see, one can use these rules for finding an optimal solution to a problem $(S, cost)$ by generating an arbitrary derivation of the form $\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots$

It will always terminate with $\ldots \implies I \parallel S' \parallel k \parallel A \implies \mathit{OptimumFound}$. Then A is a minimum-cost model for S with $cost(A) = k$. If S has no models at all, then A will be $\emptyset$ and $k = \infty$.

All the rules except Improve are natural extensions of the Abstract DPLL approach of [25]. In the following we briefly explain them.

- The Decide rule represents a case split: an undefined literal l is chosen and added to I, annotated as a decision literal.
- UnitPropagate forces a literal l to be true if there is a clause $C \vee l$ in S whose part C is false in I.
- The Optimum rule expresses that if in a state $I \parallel S \parallel k \parallel A$ in S there is a *conflicting clause* C (i.e., a clause C with $I \models \neg C$), and there is *no decision literal* in I, then the optimization procedure has terminated, which shows that the best-so-far cost is optimal.

- On the other hand, if there is a conflicting clause, and there is *at least one decision literal* in I, then one can always find (and Learn) a *backjump clause*, an entailed cost clause of the form $C \vee l' \vee c \geq k$, such that Backjump using $C \vee l'$ applies (see Lemma 1 below). Good backjump clauses can be found by *conflict analysis* of the conflicting clause [22,31], see Example 3.2 below.
- By Learn one can add any entailed cost clause to S. Learned clauses prevent repeated work in *similar* conflicts, which frequently occur in industrial problems having some regular structure. Notice that when such a clause is learned the $c \geq k$ literal is dropped (it is only kept at a meta-level for the generation of optimality certificates, see Section 4). Intuitively this is justified, since if a cost clause $C \vee c \geq k$ is entailed by $(S, cost)$, so is $C \vee c \geq k'$ for any $k' \leq k$; but as the best-so-far cost only decreases in a $\mathtt{DPLL_{BB}}$ execution, a cost clause that was entailed at a certain point of the execution will remain so at any later state $I' \parallel S' \parallel k' \parallel A'$ with respect to k'. Note also that if $S \models C$ then $(S, cost)$ entails $C \vee c \geq \infty$ and thus this Learn rule extends the usual learning mechanism of modern SAT solvers.
- Since a lemma is aimed at preventing future similar conflicts, it can be removed using Forget, when such conflicts are not very likely to be found again. In practice this is done if its *activity*, that is, how many times it has participated in *recent* conflicts, has become low.
- Restart is used to escape from bad search behaviors. The newly learned clauses will lead the heuristics for Decide to behave differently, and hopefully make $\mathtt{DPLL_{BB}}$ explore the search space in a more compact way.
- Improve updates the cost of the best model found so far. Having a stronger best-so-far cost allows one to better exploit non-trivial optimization concepts, namely *cost-based backjumping* and *cost-based propagation*.
  In a state $I \parallel S \parallel k \parallel A$, cost-based backjumping can be applied whenever $lb(I) \geq k$. This is done as follows: the lower bounding procedure can provide a *lb*-reason $C \vee c \geq k$, and, as explained above, given this conflicting clause, Backjump applies (if there is some decision literal in I; otherwise Optimum is applicable).
  A *cost-based propagation* of a literal l that is undefined in I can be made if $lb(I\,\neg l) \geq k$. Then the corresponding *lb*-reason $C \vee c \geq k$ can be learned and used to UnitPropagate l (since $I\,\neg l \models \neg C$). A form of cost-based propagation was also explained in [30]; for linear cost functions, cf. the "limit lower bound theorem" of [9].

The potential of the previous rules will be illustrated in Section 3.2. The correctness of $\mathtt{DPLL_{BB}}$ is summarized in Lemma 1 and Theorem 1.

Lemma 1 *Let $(S, cost)$ be an optimization problem, assume*

$$\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel A$$

and that there is some decision literal in I and there exists a cost clause $C \vee c \geq k$ entailed by $(S', cost)$ such that $I \models \neg C$.

Then I is of the form $I'\, l^{\mathsf{d}}\, I''$ and there exists a backjump clause, i.e., a cost clause of the form $C' \vee l' \vee c \geq k$ that is entailed by $(S', cost)$ and such that $I' \models \neg C'$ and l' is undefined in I'.

Definition 7 A derivation $\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots$ is called *progressive* if:
–it contains only finitely many consecutive Learn or Forget steps, and

–a rule is applied to any state to which some rule is applicable, and
–Restart is applied with increasing periodicity (i.e., the number of rule steps between two Restarts increases along the derivation).

Theorem 1 *Let $(S, cost)$ be an optimization problem, and consider a progressive derivation with initial state $\emptyset \parallel S \parallel \infty \parallel \emptyset$. Then this derivation is finite and of the form*

$$\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel A \implies \mathit{OptimumFound}$$

where A is a minimum-cost model for S with $cost(A) = k$. In particular, S has no models if, and only if, $k = \infty$ and $A = \emptyset$.

Of course the previous formal result provides more freedom in the strategy for applying the rules than needed. Practical implementations will only generate (progressive) derivations of a certain kind. For instance, UnitPropagate is typically applied with the highest priority, at each conflict the backjump clause is learned, and from time to time a certain portion of the learned clauses is forgotten (e.g., the 50% of less active ones). Restarts are applied with increasing periodicity by, e.g., restarting after a certain number N of conflicts and then increasing N.

In [25] detailed proofs are given (in about five pages) of a result similar to Theorem 1 for the Abstract DPLL approach that handles pure SAT, without optimization. Since the generalization of these proofs for Lemma 1 and for Theorem 1 is rather straightforward, but considerably longer and more tedious, here we sketch the main ideas of [25] and refer to the article for more details.

First several needed invariants about the states (assignments and clause sets) in derivations are proved by simple induction on the derivation length: assignments are indeed assignments (no literal will appear more than once, a literal and its negation never appear simultaneously), no new variables are introduced, the clause sets remain logically equivalent along derivations, and for each prefix of the assignment all literals are entailed by the decision literals and the clauses.

Using these invariants, a lemma similar to Lemma 1 is proved by showing how, for any state in which there is a conflicting clause and at least one decision literal, one can construct a backjump clause, i.e., an entailed clause such that Backjump using it applies. From this, it rather easily follows that any final state different from *FailState* (a tag similar to *OptimumFound* in SAT) has an assignment that is i) total, ii) that satisfies all clauses of the final clause set, and iii) is also a model of the initial clause set.

Termination of the derivations is proved using the following intuition. Consider a state more *advanced* if it has more information (literals) depending on less decisions, i.e., assignment I is more advanced than assignment I' if the prefix of I before its first decision literal is longer than in I', or it is equally long but this holds for the second decision literal, etc. This is the reason why an application of Backjump rule represents progress in the search: it adds an additional literal somewhere before the last decision. This idea leads to a well-founded lexicographic ordering on the partial assignments, and it is not hard to see that all rules make progress in this sense, except Learn and Forget (which do not change the assignment and hence it suffices to forbid infinite subsequences of them), and Restart (which therefore needs the increasing periodicity requirement).

3.2 $\mathtt{DPLL}_{\mathtt{BB}}$ Example

Consider the clause set S defined over $x_1, \ldots x_6$ (denoting $\neg x_i$ by $\bar{x}_i$):

1.	$x_2 \vee x_4$	5.	$x_1 \vee x_3 \vee \bar{x}_6$
2.	$x_2 \vee \bar{x}_5$	6.	$\bar{x}_1 \vee x_3 \vee \bar{x}_6$
3.	$x_4 \vee \bar{x}_5$	7.	$x_2 \vee x_3 \vee x_5 \vee \bar{x}_6$
4.	$x_5 \vee x_6$	8.	$x_2 \vee \bar{x}_3 \vee x_5 \vee \bar{x}_6$

where $cost(x_1, \ldots x_6) = 1x_1 + 2x_2 + \ldots + 6x_6$. We start a $\mathtt{DPLL}_{\mathtt{BB}}$ derivation, first deciding x_6 to be false (setting high-cost variables to false can be a good heuristic):

$$
\begin{array}{lll}
 & \emptyset & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} & \| S \ \| \infty \ \| \emptyset \\
\Longrightarrow_{\mathsf{Decide}} & \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S \ \| \infty \ \| \emptyset
\end{array}
$$

Now, since $\bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1$ is a model of S of cost $11 < \infty$, we can apply $\mathsf{Improve}$ and the corresponding lb-reason, e.g., $\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11$, then becomes a conflicting clause. Intuitively, it expresses that any assignment where x_2, x_4 and x_5 are set to true must have cost at least 11. Now, a *conflict analysis* procedure starting from this conflicting clause can be used to compute a backjump clause. This is done by successive resolution steps on the conflicting clause, resolving away the literals $\bar{x}_4$ and $\bar{x}_2$ in the reverse order their negations were propagated, with the respective clauses that caused the propagations:

$$
\dfrac{\dfrac{\bar{x}_2 \vee \bar{\mathbf{x}}_{\mathbf{4}} \vee \bar{x}_5 \vee c \geq 11 \quad \mathbf{x_4} \vee \bar{x}_5}{\bar{\mathbf{x}}_{\mathbf{2}} \vee \bar{x}_5 \vee c \geq 11} \qquad \mathbf{x_2} \vee \bar{x}_5}{\bar{x}_5 \vee c \geq 11}
$$

until a single literal of the current decision level (called the *1UIP*) is left, yielding $\bar{x}_5 \vee c \geq 11$. Learning the clause $C = \bar{x}_5$ allows one to jump from decision level 3 back to decision level 0 and assert x_5. All this can be modeled as follows:

$$
\begin{array}{lll}
\ldots & & \\
 & \Longrightarrow_{\mathsf{Improve}} \ \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
 & \Longrightarrow_{\mathsf{Learn}} \ \bar{x}_6^{\mathsf{d}} x_5 x_2 x_4 \bar{x}_3^{\mathsf{d}} \bar{x}_1^{\mathsf{d}} & \| S, C \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
 & \Longrightarrow_{\mathsf{Backjump}} \ \bar{x}_5 & \| S, C \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1
\end{array}
$$

Now the derivation could continue, e.g., as follows:

$$
\begin{array}{lll}
\ldots & & \\
 & \Longrightarrow_{\mathsf{UnitPropagate}} \ \bar{x}_5 x_6 & \| S, C \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
 & \Longrightarrow_{\mathsf{Decide}} \ \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} & \| S, C \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\
 & \Longrightarrow_{\mathsf{UnitPropagate}} \ \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 & \| S, C \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1
\end{array}
$$

Now notice that x_3 is not assigned, and that since x_2 and x_6 are true in the current partial assignment any assignment strictly improving the best-so-far cost 11 must assign x_3 to false. As explained above, this cost-based propagation can be modeled as follows. The lower bounding procedure expresses the fact that any solution setting x_2, x_3 and x_6 to true has cost no better than 11 by means of the lb-reason $\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11$.

This is an entailed cost clause that is learned as $C' = \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6$. Then literal $\bar{x}_3$ is propagated.

$$\begin{array}{lll} \ldots & \Longrightarrow_{\mathsf{Learn}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \quad \| S, C, C' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 \ \| S, C, C' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \end{array}$$

If we now UnitPropagate x_1 with clause 5, clause 6 becomes conflicting. As usual, a backjump clause is computed by doing conflict analysis from the falsified clause, using among others the clause C' that was learned to propagate $\bar{x}_3$:

$$\dfrac{\dfrac{\mathbf{x_1} \vee x_3 \vee \bar{x}_6 \quad \bar{\mathbf{x}}_\mathbf{1} \vee x_3 \vee \bar{x}_6}{\mathbf{x_3} \vee \bar{x}_6} \qquad \bar{x}_2 \vee \bar{\mathbf{x}}_\mathbf{3} \vee \bar{x}_6 \vee c \geq 11}{\bar{x}_2 \vee \bar{x}_6 \vee c \geq 11}$$

Learning $C'' = \bar{x}_2 \vee \bar{x}_6$ allows one to jump back to decision level 0 asserting $\bar{x}_2$.

$$\begin{array}{lll} \ldots & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 x_1 \ \| S, C, C' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Learn}} & \bar{x}_5 x_6 \bar{x}_4^{\mathsf{d}} x_2 \bar{x}_3 x_1 \ \| S, C, C', C'' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Backjump}} & \bar{x}_5 x_6 \bar{x}_2 \quad \| S, C, C', C'' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \end{array}$$

Finally after unit propagating with clause 7 one gets a conflict with clause 8, and as no decision literals are left, the optimization procedure terminates:

$$\begin{array}{lll} \ldots & \Longrightarrow_{\mathsf{UnitPropagate}} & \bar{x}_5 x_6 \bar{x}_2 x_3 \quad \| S, C, C', C'' \ \| 11 \ \| \bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1 \\ & \Longrightarrow_{\mathsf{Optimum}} & \mathit{OptimumFound} \end{array}$$

□

4 Certificates of Optimality

In the following, we show how from a certain trace of a $\mathtt{DPLL}_{\mathtt{BB}}$ execution one can extract a formal proof of optimality in a proof system asserting "A is an optimal model of S with respect to *cost*". Our proof system relies on the following type of resolution over cost clauses,

Definition 8 The *Cost Resolution* rule is the following inference rule with two cost clauses as premises and another cost clause as conclusion:

$$\dfrac{x \vee C \vee c \geq k \qquad \neg x \vee D \vee c \geq k'}{C \vee D \vee c \geq min(k, k')} \qquad \textit{Cost Resolution}$$

Cost Resolution behaves like classical resolution, except in that it further exploits the fact that $c \geq k \vee c \geq k'$ is equivalent to $c \geq min(k, k')$. In what follows, when needed a clause C from S will be seen as the trivially entailed cost clause $C \vee c \geq \infty$.

Theorem 2 *Cost Resolution is* correct*, that is, if $x \vee C \vee c \geq k$ and $\neg x \vee D \vee c \geq k'$ are cost clauses entailed by an optimization problem $(S, cost)$, then $C \vee D \vee c \geq min(k, k')$ is also entailed by $(S, cost)$.*

Proof Let $x \vee C \vee c \geq k$ and $\neg x \vee D \vee c \geq k'$ be cost clauses entailed by an optimization problem $(S, cost)$. In order to prove that $C \vee D \vee c \geq min(k, k')$ is also entailed by $(S, cost)$, let A be a model of S such that $A \models \neg(C \vee D)$, i.e., $A \models \neg C \wedge \neg D$. Thus $A \models \neg C$ and $A \models \neg D$. Now let us distinguish two cases. If $A \models \neg x$, then $A \models \neg x \wedge \neg C$, i.e., $A \models \neg(x \vee C)$. Since $x \vee C \vee c \geq k$ is entailed by $(S, cost)$, we have that $cost(A) \geq k$. Similarly, if $A \models x$, then $A \models x \wedge \neg D$, i.e., $A \models \neg(\neg x \vee D)$. As $\neg x \vee D \vee c \geq k'$ is entailed by $(S, cost)$, finally $cost(A) \geq k'$. In any case $cost(A) \geq min(k, k')$.

Definition 9 Let S be a set of cost clauses and let C be a cost clause. A *Cost Resolution proof* of C from S is a binary tree where:

- each node is (labeled by) a cost clause
- the root is C
- the leaves are clauses from S
- every non-leaf node has two parents from which it can be obtained in one Cost Resolution step.

Together with a model A such that $cost(A) = k$, a *k-lower-bound certificate* as we define now gives a precise k-optimality certificate for $(S, cost)$:

Definition 10 A *k-lower-bound certificate* for an optimization problem $(S, cost)$ consists of the following three components:

1. a set of cost clauses S'
2. a Cost-Resolution Proof of the clause $c \geq k$ from $S \cup S'$
3. for each cost clause in S', a proof of entailment of it from $(S, cost)$

As we will see, the set of cost clauses S' of component 1. of this definition corresponds to the different *lb*-reasons generated by the lower bounding procedure that may have been used along the $\texttt{DPLL}_{\texttt{BB}}$ derivation. If the lower bounding procedure is trusted, i.e., it is assumed that all cost clauses of S' are entailed, a simple independent k-lower-bound certificate checker would only need to check the cost resolution proof. Then, since by correctness of Cost Resolution the root $c \geq k$ of a Cost Resolution proof is entailed if the leaves are entailed, a k-lower-bound certificate guarantees that $c \geq k$ is indeed entailed by $(S \cup S', cost)$, and the entailment of $c \geq k$ by definition means that "$cost(A) \geq k$ for every model A of S".

If one cannot trust the lower bounding procedure, then also component 3. is needed. The notion of a "proof of entailment" from $(S, cost)$ for each cost clause in S' of course necessarily depends on the particular lower bounding procedure used, and an independent optimality proof checker should hence have some knowledge of the deductive process used by the lower bounding procedure. This aspect is addressed in detail in Section 5.

4.1 Generation of k-lower-bound certificates

Each time an *lb*-reason is generated and used in a $\texttt{DPLL}_{\texttt{BB}}$ execution, it is written to a file which we will call S' as in component 1. of the k-lower-bound certificate. Now observe that any execution of $\texttt{DPLL}_{\texttt{BB}}$ terminates with a step of Optimum, i.e., with a conflict at decision level 0. From a standard SAT solver point of view, this means that $S \cup S'$ forms an unsatisfiable SAT instance and a refutation proof for this contradiction can be reconstructed as follows (cf. [32] for details). All clauses in S and in S' get a unique identifier (ID). Each time a backjump step takes place, the backjump clause also gets a (unique) ID and a line `ID ID1 ... IDm` is written to a trace file, where `ID1 ... IDm` are the ID's of all parent clauses in the conflict analysis process generating this backjump clause. A last line is written when the conflict at decision level 0 is detected for the parents of this last conflict analysis which produces the empty clause. By processing backwards this trace file, composing all the component resolution proofs

from each conflict analysis, a resolution proof from $S \cup S'$ of the last derived clause, i.e., the empty clause, can be constructed.

If we recover the cost literals of cost clauses (recall that the Learn rule of $\mathtt{DPLL_{BB}}$ drops the cost literal) in the refutation proof, it turns out that it becomes a k-lower-bound certificate where k is the optimum of the problem. The reason is that in a Cost Resolution proof the cost literal of the root clause is the minimum among the cost literals of the leaf clauses. The following example illustrates the whole process.

Example 1 For the $\mathtt{DPLL_{BB}}$ derivation of Section 3.2, the initial clauses have ID's `1-8`, the set S' will contain the *lb*-reasons $\bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11$ and $\bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11$ with ID's `9` and `10` respectively. The two backjump clauses $\bar{x}_5 \vee c \geq 11$ and $\bar{x}_2 \vee \bar{x}_6 \vee c \geq 11$ and the final "empty" clause $c \geq 11$ get ID's `11,12,13` respectively, and the trace file will be:

$$\begin{aligned} &\mathtt{11} \leftarrow \mathtt{2}, \mathtt{3}, \mathtt{9} \\ &\mathtt{12} \leftarrow \mathtt{5}, \mathtt{6}, \mathtt{10} \\ &\mathtt{13} \leftarrow \mathtt{4}, \mathtt{7}, \mathtt{8}, \mathtt{11}, \mathtt{12} \end{aligned}$$

By processing this file backwards it is straightforward to produce a Cost Resolution proof of $c \geq 11$. This is done below, where for lack of space the proof has been split in two at the clause marked with $(*)$, and clauses have been labeled with their ID's for the sake of clarity. This proof, together with each *lb*-reason and its entailment certificate, will constitute an 11-lower-bound certificate. The optimality certificate is finally obtained with the addition of the model $\bar{x}_6 x_5 x_2 x_4 \bar{x}_3 \bar{x}_1$, which has cost 11.

$$\dfrac{\dfrac{\mathtt{7}: x_2 \vee x_3 \vee x_5 \vee \bar{x}_6 \quad \mathtt{8}: x_2 \vee \bar{x}_3 \vee x_5 \vee \bar{x}_6}{x_2 \vee x_5 \vee \bar{x}_6} \quad \dfrac{\mathtt{5}: x_1 \vee x_3 \vee \bar{x}_6 \quad \mathtt{6}: \bar{x}_1 \vee x_3 \vee \bar{x}_6}{x_3 \vee \bar{x}_6} \quad \dfrac{\mathtt{10}: \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_6 \vee c \geq 11}{\mathtt{12}: \bar{x}_2 \vee \bar{x}_6 \vee c \geq 11}}{x_5 \vee \bar{x}_6 \vee c \geq 11\ (*)}$$

$$\dfrac{\dfrac{\mathtt{4}: x_5 \vee x_6 \quad x_5 \vee \bar{x}_6 \vee c \geq 11\ (*)}{x_5 \vee c \geq 11} \quad \dfrac{\dfrac{\mathtt{9}: \bar{x}_2 \vee \bar{x}_4 \vee \bar{x}_5 \vee c \geq 11 \quad \mathtt{3}: x_4 \vee \bar{x}_5}{\bar{x}_2 \vee \bar{x}_5 \vee c \geq 11} \quad \mathtt{2}: x_2 \vee \bar{x}_5}{\mathtt{11}: \bar{x}_5 \vee c \geq 11}}{\mathtt{13}: c \geq 11}$$ □

5 Instances of Lower Bounding Procedures

In this section we show, for different classes of cost functions of increasing generality, several lower bounding procedures together with ways for proving the entailment from $(S, cost)$ for any *lb*-reason $C \vee c \geq k$ they generate.

Of course a general approach for this is to provide a list of all the models $A_1, \ldots, A_m$ of $S \wedge \neg C$, checking that each one of them has cost at least k, together with a resolution refutation of $S \wedge \neg C \wedge \neg A_1 \wedge \cdots \neg A_m$, which shows that these $A_1, \ldots, A_m$ are indeed all the models of $S \wedge \neg C$. But this will usually not be feasible in practice. Therefore, we now describe some lower bounding procedures producing simple and compact certificates that can be understood by *ad-hoc* proof checkers.

It is known [5] that every function $cost\colon \mathbb{B}^n \to \mathbb{R}$ has a unique multilinear polynomial representation of the form,

$$f(x_1, \ldots, x_n) = \sum_{Y \subseteq X} c_Y \prod_{x_j \in Y} x_j$$

where $X = \{x_1, \ldots, x_n\}$ and c_Y are real coefficients. Multilinear polynomials can also be represented as *posiforms*, i.e. multilinear polynomial expressions with *positive* coefficients depending on the set of literals,

$$\phi(x_1, \ldots, x_n) = \sum_{T \subseteq L} a_T \prod_{u \in T} u$$

where L is the set of literals ($\neg x =^{def} (1 - x)$) and $a_T \geq 0$ are nonnegative real coefficients. Note that if $\{u, \neg u\} \subseteq T$ for some $u \in L$, then they cancel away the term $\prod_{u \in T} u$ and it can be omitted. The size of the largest $T \subseteq L$ for which $a_T \neq 0$ is called the *degree* of the posiform.

5.1 Linear Cost Functions

A very important class of optimization problems is that with *linear* cost functions, i.e., of the form $cost(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i$ for certain $a_i \in \mathbb{R}$. Linear cost functions can be translated into linear posiforms $\phi(x_1, \ldots, x_n) = c_0 + \sum_{i=1}^{n} c_i x_i$ where for $i > 0$ the coefficient c_i, called the *cost* of variable x_i, is now non-negative.

Linear Boolean optimization has many applications, amongst others Automatic Test Pattern Generation [10], FPGA Routing, Electronic Design Automation, Graph Coloring, Artificial Intelligence Planning [13] and Electronic Commerce [27]. In particular the case where $c_i = 1$ for all $1 \leq i \leq n$, called the *Min-Ones* problem, appears naturally in the optimization versions of important well-known NP-complete problems such as the maximum clique or the minimum hitting set problems.

The problem of computing lower bounds for linear optimization problems in a branch-and-bound setting has been widely studied in the literature. Here we consider the two main techniques for that purpose: *independent sets* and *linear programming*.

5.1.1 Independent Sets

Given a partial assignment I and a clause C, let $undef_I(C)$ denote the set of literals in C which are undefined in I, i.e., $undef_I(C) = \{l \in C \mid l \notin I \text{ and } \neg l \notin I\}$. A set of clauses M is an *independent set for* I if:

- for all $C \in M$, neither $I \models C$ nor $I \models \neg C$ (note that this implies $undef_I(C) \neq \emptyset$);
- for all $C \in M$, $undef_I(C)$ only contains positive literals;
- for all $C, C' \in M$ such that $C \neq C'$, $undef_I(C) \cap undef_I(C') = \emptyset$.

If M is an independent set for I, any total assignment extending I and satisfying M has cost at least

$$K = c_0 + \sum_{x_i \in I} c_i + \sum_{C \in M} \min\{c_j \mid x_j \in C \text{ and } \neg x_j \notin I\}$$

since satisfying each clause C of M will require to add the minimum cost of the positive non-false (in I) literals in C. Independent sets have been used in e.g., [8,20]. In [11] they are precomputed in order to speed up the actual branch-and-bound procedure.

In this case the lower bounding procedure generates the *lb*-reason $\neg I' \vee c \geq K$, where $I' \subseteq I$ contains:

- the positive literals in I with non-null cost;
- the positive literals whose negations appear in M (which belong to I); and
- the negative literals $\neg x_i \in I$ such that $x_i \in C$ for some $C \in M$ and $c_i < \min\{c_j \mid x_j \in C \text{ and } \neg x_j \notin I\}$.
- the negative literals $\neg x_i \in I$ such that there exist $C, C' \in M$ satisfying $C \neq C'$ and $x_i \in C \cap C'$.

For this lower bounding procedure a proof of entailment of the *lb*-reason must of course contain the independent set M itself. Then the proof checker can check that $M \subseteq S$, that M is indeed independent for I and that $K \geq k$.

Example 2 Consider the clause set $S = \{ x_1 \vee x_3 \vee x_5,\ x_2 \vee x_4 \vee x_5 \vee \neg x_6,\ \neg x_1 \vee \neg x_2 \}$, and the function $cost(x_1, \ldots, x_6) = \sum_{i=1}^{6} i \cdot x_i$. We have $M = \{ x_1 \vee x_3 \vee x_5,\ x_2 \vee x_4 \vee x_5 \vee \neg x_6 \}$ is independent for the partial assignment $I = \{\neg x_5, x_6\}$. The lower bound is $6 + \min(1, 3) + \min(2, 4) = 9$, and the *lb*-reason $x_5 \vee \neg x_6 \vee c \geq 9$ is obtained. □

5.1.2 Linear Programming

This approach for computing lower bounds [18,17] relies on the fact that linear Boolean optimization is a particular case of 0-1 Integer Linear Programming. Indeed, taking into account that $\neg x = 1 - x$ for any Boolean variable x, such a Boolean optimization problem can be transformed into an integer program by transforming each clause C into the linear constraint $\sum_{l \in C} l \geq 1$. The current partial assignment I is encoded by imposing additional constraints $x = 1$ if $x \in I$, $x = 0$ if $\neg x \in I$. Then a lower bound can be computed by dropping the integrality condition and solving the resulting relaxation in the rationals with an LP solver.

If K is the lower bound obtained after solving the relaxation, an *lb*-reason of the form $\neg I' \vee c \geq K$ where $I' \subseteq I$ can be computed using an exact optimal dual solution of multipliers [28]. Namely, for each positive literal $x \in I$, if the constraint $x \geq 1$ has a non-null multiplier in the dual solution, then x is included in I'. Similarly, for each negative literal $\neg x \in I$, if the constraint $x \leq 0$ has a non-null multiplier in the dual solution, then $\neg x$ is included in I'.

Notice that in the simplex method, which is the most common algorithm for solving linear programs, an optimal dual solution is produced as a byproduct if the optimum is found. Thus an exact optimal dual solution can be obtained by means of an exact simplex-based LP solver [1], or also by recomputing the optimal dual solution in exact arithmetic using the optimal configuration of basic and non-basic variables produced by an inexact solver [14].

As for a proof of entailment of the computed *lb*-reason, the dual solution itself can be used for that purpose, as it proves the optimality of K.

Example 3 Consider again the clause set, the cost function and the partial assignment as in Example 2. In this case the linear program is

$$\begin{array}{ll} \min & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \\ \\ & \begin{array}{r} x_1 + x_3 + x_5 \geq 1 \\ x_2 + x_4 + x_5 - x_6 \geq 0 \\ -x_1 - x_2 \geq -1 \\ x_5 = 0 \\ x_6 = 1 \\ 0 \leq x_1, x_2, x_3, x_4 \leq 1 \end{array} \end{array}$$

whose optimum is 11. An optimal dual solution is $(3, 4, 2, -2, 10)$, and a proof of optimality of the lower bound is:

$$\begin{array}{lrrrrrrrrrrrrr} & x_1 & +2\,x_2 & +3\,x_3 & +4\,x_4 & +5\,x_5 & +6\,x_6 & -11 & = \\ +\,\mathbf{3}\ (& x_1 & & +\ x_3 & & +\ x_5 & & -1 &) \\ +\,\mathbf{4}\ (& & x_2 & & +\ x_4 & +\ x_5 & -\ x_6 & &) \\ +\,\mathbf{2}\ (& -\,x_1 & -\ x_2 & & & & & +1 &) \\ -\,\mathbf{2} & & & & & x_5 & & & \\ +\,\mathbf{10}\ (& & & & & & x_6 & -1 &) \end{array}$$

This proves that $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 + 6x_6 \geq 11$ for all x_1, x_2, x_3, x_4, x_5, x_6 such that $x_1 + x_3 + x_5 \geq 1$, $x_2 + x_4 + x_5 - x_6 \geq 0$, $-x_1 - x_2 \geq -1$, $x_5 \leq 0$ and $x_6 \geq 1$, and so can be used as a proof of entailment of the *lb*-reason $x_5 \vee \neg x_6 \vee c \geq 11$ (notice that none of the literals of the assignment is dropped in the *lb*-reason since both $x_5 \leq 0$ and $x_6 \geq 1$ are used). □

Example 4 Consider the clause set $S = \{ \neg x_1 \vee x_2 \vee x_3,\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \}$, the cost function $cost(x_1, x_2, x_3) = x_2 + x_3$ and the partial assignment $I = \{x_1, x_2, \neg x_3\}$. The corresponding relaxation is

$$\begin{array}{ll} \min & x_2 + x_3 \\ \\ & \begin{array}{r} -x_1 + x_2 + x_3 \geq 0 \\ -x_1 - x_2 - x_3 \geq -2 \\ x_1 = 1 \\ x_2 = 1 \\ x_3 = 0 \end{array} \end{array}$$

In this case the minimum is 1, as proved by the optimal dual solution $(1, 0, 1, 0, 0)$:

$$\begin{array}{lrrrr} & & x_2 & +\ x_3 & -1 & = \\ +\,\mathbf{1}\ (& -\,x_1 & +\ x_2 & +\ x_3 & &) \\ +\,\mathbf{1}\ (& x_1 & & & -1 &) \end{array}$$

This shows that $x_2 + x_3 \geq 1$ for all x_1, x_2, x_3 such that $-x_1 + x_2 + x_3 \geq 0$ and $x_1 \geq 1$. Thus an *lb*-reason for the lower bound 1 is $\neg x_1 \vee c \geq 1$, and the above proof is a certificate. Note that this *lb*-reason is shorter than $\neg x_1 \vee \neg x_2 \vee x_3 \vee c \geq 1$, which is the clause obtained by taking the literals of the partial assignment that evaluate to 0 in the active constraints of the optimal solution [21]. □

5.2 Quadratic Cost Functions

Another important class of optimization problems is that of *quadratic* cost functions, i.e., of the form $cost(x_1, \ldots, x_n) = \sum_{i=1}^{n} c_i x_i + \sum_{1 \leq i < j \leq n} c_{ij} x_i x_j$. Quadratic cost functions can be translated into quadratic posiform such as,

$$\phi(x_1, \ldots, x_n) = a_0 + \sum_{u \in L} a_u u + \sum_{u,v \in L} a_{uv} uv$$

where, as before, a_0, a_u and a_{uv} are positive coefficients[2]. The problem of computing lower bounds for quadratic posiforms has also attracted a lot of attention, especially in the Operations Research field. Here we consider the most widely used technique, roof-duality bound [12], and its efficient flow-based computation [5].

Let us first introduce an additional dummy variable x_0 for which we will always assume value 1. Using this new variable, we can rewrite the previous posiform in its *homogeneous representation* (i.e., without linear terms),

$$\phi(x_0, x_1, \ldots, x_n) - a_0 = \sum_{u,v \in L'} a_{uv} uv$$

The set $L' = L \cup \{x_0, \neg x_0\}$ is the new set of literals. The new coefficients are defined as follows: $a_{x_0 v} = a_v$ and $a_{\bar{x}_0 v} = 0$.

From the previous posiform, we obtain a capacitated network G_ϕ as follows. For each literal in L' there is a node in G_ϕ. For each non-zero coefficient a_{uv} there are two arcs $(u, \neg v)$ and $(v, \neg u)$. The capacity of arc (u, v) is $\frac{1}{2} a_{u\bar{v}}$. Observe that this network is reminiscent of the so-called *implication graph* defined for 2-SAT formulas [4]. The connection becomes clear if we think of the binary terms $a_{uv}uv$ as binary clauses $\neg u \vee \neg v$.

Let φ be a maximum feasible flow of G_ϕ from source x_0 to sink $\neg x_0$. The *value* of φ (i.e., the amount of flow departing from the source or, equivalently, arriving to the sink) is the so-called roof-dual lower bound of the posiform optimum. The intuition behind this lower bound is that each path from the source to the sink reflects a set of terms in the posiform that cannot be simultaneously assigned without incurring in a positive cost.

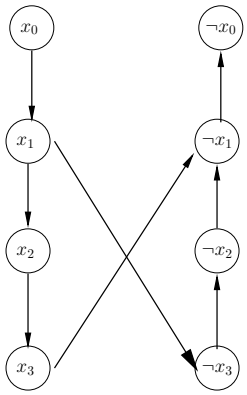
Consider now a branch and bound execution. Let I be a partial assignment and let ϕ_I denote the posiform obtained when the variables defined by I are accordingly instantiated in ϕ. We say that a literal $l \in I$ is *relevant with respect to* ϕ if it appears in ϕ.

Let G_{ϕ_I} be the corresponding capacitated network and let K be the value of a maximum flow of G_{ϕ_I}, which is a lower bound of ϕ_I. In this case, the lower bounding procedure generates the *lb*-reason $\neg I' \vee c \geq K$, where $I' \subseteq I$ contains those literals in I that are relevant with respect to ϕ.

For this lower bounding procedure a proof of entailment of the *lb*-reason is the maximal flow itself. The proof checker can check that the maximal flow is indeed a flow of $G_{\phi'_I}$ and that its value is K.

[2] The transformation is achieved by replacing each negative quadratic term $c_{ij}x_i x_j$ by $c_{ij}x_i - c_{ij}x_i\bar{x}_j$ and subsequently each negative linear term $c_i x_i$ by $c_i - c_i\bar{x}_i$.

Example 5 Consider the cost function $cost(x_1, \ldots, x_5) = x_1 + x_2 + x_4 - x_1x_2 + x_1x_3 - x_1x_4 - x_2x_3 + x_2x_5$. Its equivalent posiform is $\phi(x_1, \ldots, x_5) = x_1\bar{x}_2 + x_1x_3 + \bar{x}_1x_4 + x_2\bar{x}_3 + x_2x_5$. Let $I = \{x_4, \bar{x}_5\}$. Note that $\bar{x}_5$ is irrelevant with respect to the posiform. Then $\phi_I(x_1, x_2, x_3) = \bar{x}_1 + x_1\bar{x}_2 + x_1x_3 + x_2\bar{x}_3$. Its homogeneous representation is $\phi_I(x_0, x_1, x_2, x_3) = x_0\bar{x}_1 + x_1\bar{x}_2 + x_1x_3 + x_2\bar{x}_3$ and the associated capacitated network G_{ϕ_I},



All capacities are 0.5 and are omitted in the previous drawing. It is easy to check that there are only two paths from x_0 to $\neg x_0$. Then the possible maximum flows can only go along these two paths and have cost 0.5. Consequently, the roof-dual bound is 0.5. In this particular case we can do slightly better and give the bound $\lceil 0.5 \rceil = 1$ because all the coefficients of the cost function are integers and therefore it always returns integer values. The lower bounding procedure generates the *lb*-reason $\neg x_4 \vee c \geq 1$. □

5.3 Cost Functions of Arbitrary Degree

In this subsection we consider the most general case of cost functions being multilinear polynomials of arbitrary degree which, in turn, can always be represented as posiforms of arbitrary degree.

Our first observation is that this problem is equivalent to (partial weighted) Max-SAT, where the cost function is defined by a set of so-called *soft* clauses $C \in S'$ with a *weight* function $\omega_C : S' \to \mathbb{R}$. Then the cost of a total assignment A is the sum of the weights of the clauses in S' that are false in A. It is easy to realize that similar costs can be obtained with the following posiform,

$$\phi(x_1, \ldots, x_n) = \sum_{C \in S'} (w_C \prod_{l \in C} \neg l)$$

Note that S' is disjoint from the (possibly empty) set of clauses S, which are called *hard* clauses in this context. Thus, the corresponding optimization problem is (S, ϕ).

Max-SAT has many applications, among others Probabilistic Reasoning [26], Frequency Assignment [6], Computer Vision, Machine Learning and Pattern Recognition (see the introduction of [29]).

Note that Max-SAT can be expressed as a linear optimization problem by adding slack variables to soft clauses and taking the weighted sum of these variables as the cost function [2]. However, this translation is normally impractical, making the SAT solver extremely slow, since, e.g., it hinders the application of unit propagation [3].

Most of the research in recent years in the Max-SAT community has been devoted to the computation of good quality lower bounds to be used within a branch-and-bound setting. As shown in [15], most of these lower bounding procedures can be seen as limited forms of *Max-resolution* (see below). Since Max-resolution is sound, theoretically one can in fact use it to certify optimality in any Max-SAT problem. But the growth in the number of clauses makes this impractical except for small problems. However, one can use it for the proof of entailment for individual *lb*-reasons.

For simplicity, we show here Max-resolution for soft clauses of the form $(l_1 \vee l_2, w)$, where w denotes the weight:

$$\frac{(x \vee a, u) \quad (\neg x \vee b, v)}{(a \vee b, m)(x \vee a, u - m)(\neg x \vee b, v - m)(x \vee a \vee \neg b, m)(\neg x \vee b \vee \neg a, m)}$$

where $m = \min(u, v)$ and the conclusions replace the premises instead of being added to the clause set.

Example 6 Consider a Max-SAT problem without hard clauses and where soft clauses are $S' = \{\ (x_1 \vee x_2 \vee x_3, 1),\ (x_1 \vee \neg x_2 \vee x_3, 2),\ (\neg x_1 \vee x_2 \vee x_3, 3),\ (\neg x_1 \vee \neg x_2 \vee x_3, 4)\ \}$. Given the partial assignment $I = \{\neg x_3, x_4\}$, by means of the following steps of Max-resolution

$$\dfrac{\dfrac{\dfrac{(x_1 \vee \mathbf{x_2} \vee x_3, 1) \quad (x_1 \vee \neg \mathbf{x_2} \vee x_3, 2)}{\vdots}}{(\mathbf{x_1} \vee x_3, 1)} \quad \dfrac{\dfrac{(\neg x_1 \vee \mathbf{x_2} \vee x_3, 3) \quad (\neg x_1 \vee \neg \mathbf{x_2} \vee x_3, 4)}{\vdots}}{(\neg \mathbf{x_1} \vee x_3, 3)}}{(x_3, 1)}$$

one gets clause x_3 with weight 1. Taking into account the partial assignment $I = \{\neg x_3, x_4\}$, this clause implies that 1 is a lower bound and an *lb*-reason is $\neg x_3 \vee c \geq 1$. Moreover, the proof of Max-resolution above proves the entailment of the *lb*-reason. □

6 Experimental Evaluation

The goal of this section is to provide empirical evidence of the feasibility of the approach for certifying Boolean optimization presented in this paper. More specifically, our results below indicate that the overhead of proof producing is only a small fraction of the solving time, and that at worst proof checking can be carried out in an amount of time comparable to the solving time.

In this experimental assessment we have focused on a particular kind of cost functions and a concrete method for obtaining lower bounds. Namely, we have addressed linear optimization problems by using LP-based lower bounding; see Section 5.1.2. We have implemented a proof-producing Boolean optimizer by equipping a proof-producing SAT solver with optimization capabilities by means of the LP solver CPLEX [14]. We have also developed the infrastructure for checking the certificates generated by the Boolean optimizer:

1. the tool TraceCheck[3], which reproduces the Cost Resolution proof from the trace file, see Section 4.1;
2. a program that, given the trace file, identifies which clauses appear in the Cost Resolution proof and checks that (i) only original clauses and *lb*-reasons appear, and (ii) the *lb*-reason with the best cost appears;
3. a lower bound certificate checker, which ensures that *lb*-reasons are entailed.

All items in the above list have been implemented on our own except for the first. In particular, the lower bound certificate checker was needed because we decided not to trust CPLEX: as explained in Section 5.1.2, optimal dual solutions are used for generating *lb*-reasons, and since CPLEX is implemented with floating-point arithmetic, these dual solutions might be subject to inaccuracies. The lower bound certificate checker recomputes the exact optimal dual solution from the optimal configuration of basic and non-basic variables produced by CPLEX, and using those multipliers checks that the *lb*-reasons are entailed; see Examples 4 and 5.

The benchmarks for the experiments reported here come from the family "logic-synthesis" of the INDUSTRIAL, OPT-SMALLINT category of the Pseudo-Boolean Evaluation 2006 [4]. These instances were chosen as it was known that LP-based lower bounding was effective on them [18,17].

Figure 1 shows the results of our experiments. The measurements were performed on a PC with an Intel Pentium 4 CPU clocked at 3.4 GHz, equipped with 1.5 GB of RAM and running Ubuntu Linux. The timeout was set to 1800 seconds, the same as in the Pseudo Boolean Evaluations. The description of the columns is as follows. The first column (**Name**) shows the name of each instance. The second column (**S**) reports on solving time with proof production disabled, while the third column (**P**) shows the difference between solving time with and without proof production, that is, the overhead of proof production. The fourth column (%**P**/**S**) shows the percentage of this overhead over solving time. The rest of the columns measure the cost of checking the certificates. The fifth (**R**) and sixth (**LB**) columns are the time spent on the Cost Resolution proof (items 1 and 2 in the list above) and checking lower bound certificates (item 3), respectively. The seventh column (**C**) is the total time spent on checking, i.e., the addition of the previous two columns. Finally the last column (%**C**/**S**) shows the percentage of time spent on checking over time spent on solving. Timings are measured in seconds.

Interestingly enough, for all those instances that were solved (without proof production) within the time limit the solutions could be checked to be optimal, and this was done within the same time limit too (instances that timed out at solving stage have not been shown in Figure 1). Moreover, except for easy problems that were solved in less than 1 second, the overhead of proof producing was at most 25% of the solving time. As regards proof checking, on average the percentage of checking time over solving time is about 50%, and on most cases it is smaller. On the other hand, for some particular cases this ratio is higher. One of the reasons for this is that our lower bound certificate checker uses infinite-precision arithmetic, which is less efficient than computing in floating-point arithmetic, as done in CPLEX. Furthermore, note that our implementation is a preliminary prototype, since our aim was to show the feasibility of our approach for certifying Boolean optimization, and not to develop a finely-tuned

[3] Available at `http://fmv.jku.at/tracecheck`

[4] These benchmarks are available at `http://www.cril.univ-artois.fr/PB07`

Name	S	P	%P/S	R	LB	C	%C/S
bbara.r	0.03	0.02	66	0.02	0.01	0.03	100
ex5inp.r	0.04	0.04	100	0.02	0.02	0.04	100
m50_100_10_10.r	0.33	0.13	39	0.06	0.13	0.19	58
m50_100_10_15.r	1.69	0.37	22	0.01	0.55	0.56	33
m100_50_10_10.r	5.80	0.7	12	0.08	4.03	4.11	71
m100_100_10_10.r	30.79	7.77	25	0.51	18.88	19.39	63
opus.r	0.02	0.03	150	0	0.01	0.01	50
ex6inp.r	0.85	0.21	25	0	0.10	0.1	12
m100_100_10_15.r	39.58	4.97	13	0.82	25.23	26.05	66
m100_300_10_10.r	88.71	19.19	22	1.16	42.17	43.33	49
m50_100_30_30.r	11.69	2.51	21	0.40	3.22	3.62	31
m100_300_10_14.r	48.73	11.25	23	0.62	25.27	25.89	53
m100_300_10_15.r	241.63	53.73	22	4.47	137.52	141.99	59
m100_300_10_20.r	176.60	36.01	20	2.11	104.20	106.31	60
m100_50_20_20.r	9.23	1.22	13	0.45	3.75	4.2	46
m100_100_10_30.r	11.01	1.54	14	0.17	6.25	6.42	58
dk512x.r	0.18	0.04	22	0	0.03	0.03	17
maincont.r	0.12	0.05	42	0	0.02	0.02	17
m50_100_50_50.r	4.06	0.46	11	0.04	0.69	0.73	18
fout.r	4.00	0.57	14	0	0.31	0.31	8
m100_50_30_30.r	2.10	0.04	2	0	0.45	0.45	21
m100_100_30_30.r	66.79	7.17	11	1.49	39.39	40.88	61
m50_100_70_70.r	0.78	0.09	12	0	0.08	0.08	10
C880.a	0.70	0.37	53	0	0.35	0.35	50
m100_50_40_40.r	0.32	0.03	9	0	0.04	0.04	12
m50_100_90_90.r	1.35	0.03	2	0	0.02	0.02	1
mlp4.r	29.97	2.04	7	0.03	1.37	1.4	5
m100_100_50_50.r	162.49	20.99	13	25.85	38.33	64.18	39
max512.r	20.42	1.28	6	0.01	0.64	0.65	3
m100_100_70_70.r	25.79	1.15	4	0.12	1.44	1.56	6
exps.r	17.67	0.61	3	0	0.37	0.37	2
addm4.r	80.62	2.3	3	0.02	1.78	1.8	2
test1.r	276.25	32.6	12	0.19	19.25	19.44	7
m100_100_90_90.r	3.58	0.11	3	0	0.02	0.02	1
max1024.pi	579.86	67.75	12	0.75	716.92	717.67	120
max1024.r	677.52	150.7	22	0.86	728.58	729.44	110
ex4inp.r	4.07	0.51	13	0	0.19	0.19	5
m4.r	38.38	1.98	5	0.01	1.25	1.26	3
lin.rom.r	200.80	13.55	7	0.13	30.69	30.82	15
rd73.b	1.99	0.12	6	0	1.33	1.33	67
ricks.r	45.90	2.85	6	0.04	1.52	1.56	3
sao2.b	31.43	5.6	18	0.10	17.67	17.77	57
f51m.b	28.56	5.63	20	0.21	9.62	9.83	34
clip.b	2.75	0.4	15	0	0.44	0.44	16
count.b	231.52	24.55	11	1.02	223.42	224.44	97
C880.b	322.45	38.48	12	0.77	1230.38	1231.15	380
9sym.b	1.78	0.16	9	0	2.69	2.69	150
jac3	51.75	2.96	6	0.04	7.43	7.47	14
5xp1.b	108.84	7.49	7	0.14	45.31	45.45	42
ex5.r	1280.80	143.3	11	6.35	543.11	549.46	43

Fig. 1 Table comparing solving time with proof production overhead and proof checking time.

system. In particular, the procedures in our prototype for factorizing matrices, which are called when recomputing dual solutions, have not been as carefully implemented as those in CPLEX, which has a significant effect on some instances.

7 Extensions

To show the flexibility of our framework, in this section we briefly outline two interesting extensions of it: for computing the m *best* (i.e., lowest-cost) models for some m, and for computing *all models with cost lower than a certain threshold* k. We show that also the certificates for these can be derived without much effort.

7.1 Finding the m best models

For handling this extension, we consider a slight modification in $\texttt{DPLL}_{\texttt{BB}}$ where the fourth component of states becomes an ordered sequence of m models $(A_1, \ldots, A_m)$, such that $cost(A_{i-1}) \leq cost(A_i)$ for all i in $2...m$. Furthermore, the semantics of the third component k is defined by the invariant that $cost(A_m) = k$, that is, k is the cost of the worst (i.e., the one with highest cost) of the best m models found so far.

The only rule that changes is Improve, which now amounts to replacing the worst of the best so far models by a better one, becoming:

ImproveTop-m :

$$I \parallel S \parallel k \parallel (A_1, \ldots, A_m) \implies I \parallel S \cup \{\neg I\} \parallel k' \parallel (B_1, \ldots, B_m)$$

$$\textbf{if} \begin{cases} I \models S \text{ and } cost(I) < k \text{ and} \\ (B_1, \ldots, B_m) = sort(A_1, \ldots, A_{m-1}, I) \\ \text{and } cost(B_m) = k' \end{cases}$$

where *sort* indicates the sorting function from lowest (first) to highest (last) cost.

Let us briefly comment on All-SAT, that is, enumerating all models of a set of clauses. In a sense here we are doing a hybrid between branch and bound and All-SAT: instead of keeping *all* models, we only keep the m best ones found so far. Indeed, as in All-SAT, each time the ImproveTop-m rule is triggered, keeping a new model I, a blocking clause $\neg I$ is added in order to preclude repeating that same model again. Note that $\neg I$ is conflicting at that point, and hence a backjump clause can be learned and Backjump can be applied.

Similarly to what we did before, we use $\emptyset$ to denote an empty assignment, where $cost(\emptyset) = \infty$, and we get:

Theorem 3 *Let $(S, cost)$ be an optimization problem, and consider a progressive derivation with ImproveTop-m instead of Improve, with initial state $\emptyset \parallel S \parallel \infty \parallel (\emptyset, \ldots, \emptyset)$. Then this derivation is finite and of the form*

$$\emptyset \parallel S \parallel \infty \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel (A_1, \ldots, A_m) \implies \mathit{OptimumFound}$$

where $(A_1, \ldots, A_m)$ are minimal-cost models of S, that is, there is no model A of S with $A \notin \{A_1, \ldots, A_m\}$ such that $cost(A) < cost(A_m)$.

As before, from the trace of the $\texttt{DPLL}_{\texttt{BB}}$ execution one can extract a formal proof of optimality in a proof system. In this case it asserts that "$(A_1, \ldots, A_m)$ are minimum-cost models of S". The intuition for this is the following: if the derivation ends with $I \parallel S' \parallel k \parallel (A_1, \ldots, A_m) \implies \mathit{OptimumFound}$, since at each ImproveTop-m step we have added $\neg I$ as a new clause, all the A_i in $(A_1, \ldots, A_m)$ are now blocked, so now the certificate means the same as in the previous section: that there is no *other* model A of S (that is, $A \notin \{A_1, \ldots, A_m\}$), with $\mathit{cost}(A) < k = \mathit{cost}(A_m)$.

7.2 Finding all models better than a given k

This extension is very similar to the previous one. What changes is that the third component k is fixed, i.e., it does not change, and now the fourth component of states is a set of models M.

Again the only rule that changes is Improve, which is replaced by the following new rule that amounts to adding another model to M if its cost is below k:

AddModel :

$$I \parallel S \parallel k \parallel M \implies I \parallel S \cup \{\neg I\} \parallel k \parallel M \cup \{I\} \quad \textbf{if} \quad \left\{ I \models S \text{ and } \mathit{cost}(I) < k \right.$$

and we get:

Theorem 4 *Let (S, cost) be an optimization problem, and consider a progressive derivation with AddModel instead of Improve, with initial state $\emptyset \parallel S \parallel k \parallel \emptyset$. Then this derivation is finite and of the form*

$$\emptyset \parallel S \parallel k \parallel \emptyset \implies \ldots \implies I \parallel S' \parallel k \parallel M \implies \mathit{OptimumFound}$$

where M is the set of all models A of S with $\mathit{cost}(A) < k$.

As before, from the trace of the $\texttt{DPLL}_{\texttt{BB}}$ execution one can extract a formal proof of optimality in a proof system asserting "M is the set of all models A of S with $\mathit{cost}(A) < k$", and again the certificate means that there is no *other* model A of S with $\mathit{cost}(A) < k$.

8 Conclusions

Our abstract DPLL-based branch-and-bound algorithm, although being very similar to abstract DPLL, can model optimization concepts such as cost-based propagation and cost-based learning. Thus, $\texttt{DPLL}_{\texttt{BB}}$ is natural to SAT practitioners, but still faithful to most state-of-the-art branch-and-bound solvers. Interestingly, several branch-and-bound solvers, even state-of-the-art ones, still do not use cost-based backjumping and propagation, which appear naturally in $\texttt{DPLL}_{\texttt{BB}}$. Our formal definition of optimality certificates and the description of how a $\texttt{DPLL}_{\texttt{BB}}$ trace can be used to generate them turns out to be elegant and analogous to the generation of refutation proofs by resolution in SAT. We think that $\texttt{DPLL}_{\texttt{BB}}$ will help understanding and reasoning about new branch-and-bound implementations and further extensions.

Acknowledgments. We would like to thank Roberto Asín Achá for providing us with a parser for the benchmarks of the experimental evaluation.

References

1. A. Makhorin: (2007). GLPK 4.25 (GNU Linear Programming Kit). Available at `http://www.gnu.org/software/glpk/`
2. Amgoud, L., Cayrol, C., Berre, D.L.: Comparing Arguments Using Preference Ordering for Argument-Based Reasoning. In: ICTAI, pp. 400–403 (1996)
3. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Efficient Generation of Unsatisfiability Proofs and Cores in SAT. In: Proceedings of LPAR'08, *Lecture Notes in Computer Science*, vol. 5330, pp. 16–30. Springer (2008)
4. Aspvall, B., Plass, M.F., Tarjan, R.E.: A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. Information Processing Letters **8**(3), 121 – 123 (1979)
5. Boros, E., Hammer, P.L.: Pseudo-Boolean Optimization. Discrete Applied Mathematics **123**(1-3), 155–225 (2002)
6. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio Link Frequency Assignment. Constraints **4**, 79–89 (1999)
7. Cimatti, A., Griggio, A., Sebastiani, R.: A Simple and Flexible Way of Computing Small Unsatisfiable Cores in SAT Modulo Theories. In: Proceedings of SAT'07, *Lecture Notes in Computer Science*, vol. 4501, pp. 334–339. Springer (2007)
8. Coudert, O.: On Solving Binate Covering Problems. In: Proceedings of DAC'96, pp. 197–202. ACM (1996)
9. Coudert, O., Madre, J.C.: New Ideas for Solving Covering Problems. In: Proceedings of DAC'95, pp. 641–646. ACM (1995)
10. Flores, P.F., Neto, H.C., Marques-Silva, J.P.: An Exact Solution to the Minimum Size Test Pattern Problem. ACM Trans. Des. Autom. Electron. Syst. **6**(4), 629–644 (2001).
11. Fu, Z., Malik, S.: Solving the Minimum Cost Satisability Problem Using SAT Based Branch-and-Bound Search. In: Proceedings of ICCAD'06, pp. 852 – 859 (2006)
12. Hammer, P., Hansen, P., Simeone, B.: Roof duality, complementation and persistency in quadratic 0-1 optimization. Mathematical Programming **28**, 121–155 (1984)
13. Holger H. Hoos and Thomas Stützle: SATLIB: An Online Resource for Research on SAT. In: Proceedings of SAT'00, pp. 283–292. IOS Press (2000). SATLIB is available online at www.satlib.org
14. ILOG S.A.: (2007). ILOG CPLEX Version 11.000. `http://www.ilog.com/products/cplex`
15. Larrosa, J., Heras, F., de Givry, S.: A Logical Approach to Efficient Max-SAT Solving. Artif. Intell. **172**(2-3), 204–233 (2008)
16. Larrosa, J., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Branch and Bound for Boolean Optimization and the Generation of Optimality Certificates. In: 12th International Conference on Theory and Applications of Satisfiability Testing, SAT'09, Lecture Notes in Computer Science 5584, pp. 453–466 (2009)
17. Li, X.Y.: Optimization Algorithms for the Minimum-Cost Satisfiability Problem. Ph.D. thesis, Dept. Comp. Sc., N. Carolina State Univ. (2004)
18. Liao, S., Devadas, S.: Solving Covering Problems Using LPR-Based Lower Bounds. In: Procs. DAC'97, pp. 117–120. ACM (1997)
19. Manquinho, V.M., Marques-Silva, J.P.: Search Pruning Conditions for Boolean Optimization. In: Proceedings of ECAI 2000, pp. 103–107. IOS Press (2000)
20. Manquinho, V.M., Marques-Silva, J.P.: Search Pruning Techniques in SAT-Based Branch-And-Bound Algorithms for the Binate Covering Problem. IEEE Trans. on CAD of Integrated Circuits and Systems **21**(5), 505–516 (2002)
21. Manquinho, V.M., Marques-Silva, J.P.: Satisfiability-Based Algorithms for Boolean Optimization. Ann. Math. Artif. Intell. **40**(3-4), 353–372 (2004)
22. Marques-Silva, J., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
23. Moskal, M.: Rocket-Fast Proof Checking for SMT Solvers. In: 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08, *Lecture Notes in Computer Science*, vol. 4963, pp. 486–500. Springer (2008)
24. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: Proceedings of SAT'06, *Lecture Notes in Computer Science*, vol. 4121, pp. 156–169. Springer (2006)
25. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM **53**(6), 937–977 (2006)

26. Park, J.D.: Using Weighted Max-SAT Engines to Solve MPE. In: Proc. of the 18^{th} AAAI, pp. 682–687. Edmonton, Alberta, Canada (2002)
27. Sandholm, T.: An Algorithm for Optimal Winner Determination in Combinatorial Auctions. In: IJCAI-99, pp. 542–547 (1999)
28. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons, Chichester (1986)
29. Werner, T.: A Linear Programming Approach to Max-Sum Problem: A Review. Tech. Rep. CTU-CMP-2005-25, Center for Machine Perception, Czech Technical University (2005)
30. Xing, Z., Zhang, W.: Maxsolver: An Efficient Exact Algorithm for (Weighted) Maximum Satisfiability. Artificial Intelligence **164**(1-2), 47–80 (2005)
31. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: ICCAD, pp. 279–285 (2001)
32. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In: 2003 Conference on Design, Automation and Test in Europe Conference, DATE'03, pp. 10,880–10,885. IEEE Computer Society (2003)