# Solving Optimization Problems with DLL

**Enrico Giunchiglia** [1] and **Marco Maratea**[1, 2]

**Abstract.**

Propositional satisfiability (SAT) is a success story in Computer Science and Artificial Intelligence: SAT solvers are currently used to solve problems in many different application domains, including planning and formal verification. The main reason for this success is that modern SAT solvers can successfully deal with problems having millions of variables. All these solvers are based on the Davis-Logemann-Loveland procedure (DLL). DLL is a decision procedure: Given a formula $\varphi$, it returns whether $\varphi$ is satisfiable or not. Further, DLL can be easily modified in order to return an assignment satisfying $\varphi$, assuming one exists. However, in many cases it is not enough to compute a satisfying assignment: Indeed, the returned assignment has also to be "optimal" in some sense, e.g., it has to minimize/maximize a given objective function.

In this paper we show that DLL can be very easily adapted in order to solve optimization problems like MAX-SAT and MIN-ONE. In particular these problems are solved by simply imposing an ordering on a set of literals, to be followed while branching. Other popular problems, like DISTANCE-SAT and WEIGHTED-MAX-SAT, can be solved in a similar way. We implemented these ideas in ZCHAFF and the experimental analysis show that the resulting system is competitive with respect to other state-of-the-art systems.

## 1 Introduction

Propositional satisfiability (SAT) is a success story in Computer Science and Artificial Intelligence: SAT solvers are currently used to solve problems in many different application domains, including planning [11], formal verification [6], and many others such as RNA folding, hand-writing recognition, graph isomorphism and sudoku problems. The main reason for this success is that modern SAT solvers can successfully deal with problems having millions of variables [5]. All these solvers are based on the Davis-Logemann-Loveland procedure (DLL) [9]. DLL is a decision procedure: Given a formula $\varphi$, it returns whether $\varphi$ is satisfiable or not; Further, DLL can be easily modified in order to return an assignment satisfying $\varphi$, assuming one exists. However, in many cases it is not enough to compute a satisfying assignment: Indeed, the returned assignment has also to be "optimal" in some sense, e.g., it has to minimize/maximize a given objective function. MIN-ONE and MAX-SAT are two optimization versions of SAT. In MIN-ONE (resp. MIN-ONE$_\subseteq$), given a satisfiable instance, the goal is to find a satisfying assignment in which the set of variables assigned to true is of minimal cardinality (resp. minimal). In MAX-SAT (resp. MAX-SAT$_\subseteq$), given an unsatisfiable instance, the goal is to find an assignment in which the set of satisfied clauses is of maximal cardinality (resp. maximal). MIN-ONE and MAX-SAT problems have been studied, e.g., in [8, 16], and are particular cases of DISTANCE-SAT [3] and Pseudo-Boolean (see e.g., [4]) problems. Our interest for MIN-ONE, MIN-ONE$_\subseteq$, MAX-SAT and MAX-SAT$_\subseteq$ problems are related to their applications in the area of planning and formal verification, as briefly discussed in the rest of the paper. Besides these application domains, [8] shows that many important graph problems involving combinatorial optimization (such as Max-Cut, Max-Clique, and Min Vertex Cover) have linear-time reductions to MIN-ONE and MAX-SAT problems. Considering the procedures used to solve such problems, the standard approach is to modify DLL following a branch&bound schema. In the MIN-ONE case, whenever an assignment satisfying the input formula $\varphi$ and with cost $c_\mu$ is found, search continues looking for another assignment satisfying $\varphi$ but with a lower cost: The drawback of such approaches is that they may generate, at least in principle, all the assignments satisfying $\varphi$. Analogously for MAX-SAT problems.

In this paper we show that DLL can be very easily adapted in order to solve optimization problems like MIN-ONE, MAX-SAT and their variants. In particular these problems are solved by simply imposing an ordering on a set of literals, to be followed while branching. With such modification —differently from what happens for branch&bound approaches— the first discovered assignment satisfying the input formula, is guaranteed to be "optimal": This assumes we are given a MIN-ONE/MIN-ONE$_\subseteq$ problem, but analogous considerations hold for the MAX-SAT/MAX-SAT$_\subseteq$ case. Other optimization problems, like DISTANCE-SAT and WEIGHTED-MAX-SAT, can be solved in a similar way. We implemented these ideas in ZCHAFF and the experimental analysis we conducted on MIN-ONE and MAX-SAT problems shows that:

- On MIN-ONE and MAX-SAT problems, our system is competitive with respect to other dedicated solvers and state-of-the-art systems used in the last Pseudo-Boolean evaluation [12].
- Considering MIN-ONE$_\subseteq$ problems, our solver is the fastest. In particular, our solver is much faster in solving MIN-ONE$_\subseteq$ instances than the corresponding MIN-ONE instances, while this is not the case for MAX-SAT$_\subseteq$ with respect to theMAX-SAT.
- Related to the previous point, comparing the cardinality $\#C$ (resp. $\#C_\subseteq$) of the set of true variables returned when solving a MIN-ONE and a MIN-ONE$_\subseteq$ problem, we see that for most instances $\#C = \#C_\subseteq$. Comparing the analogous values for MAX-SAT and MAX-SAT$_\subseteq$, these are equal on all instances but three. Thus, provided we have an efficient solver for MIN-ONE$_\subseteq$ (resp. MAX-SAT$_\subseteq$), it makes sense to use it also for MIN-ONE (resp. MAX-SAT) problems, at least for computing a "good" upper (resp. lower) bound.

The paper is structured as follows. In Section 2, we give the basic terminology and notation. We then present OPT-DLL, i.e., DLL modified in order to solve optimization problems (Section 3). How to model and solve optimization problems with OPT-DLL is showed in Section 4. The last two sections are devoted to the experimental analysis and the conclusions.

[1] DIST - Universitá di Genova
[2] Dipartimento di Matematica - Universitá della Calabria

## 2 Formal preliminaries

Given a set $S$, a relation "$\prec \subseteq S \times S$" is a *(strict or irreflexive) partial order on* $S$ if it has the following properties:

1. *Irreflexivity*: $a \not\prec a$, for each $a \in S$.
2. *Antisymmetry*: $a \prec b$ and $b \prec a$ implies $a = b$.
3. *Transitivity*: $a \prec b$ and $b \prec c$ implies $a \prec c$.

If for each two distinct $a, b \in S$, $a \prec b$ or $b \prec a$ then $\prec$ is said to be a *total order*. It is common to call the pair $S, \prec$ a *partially ordered set*.

Consider a set $P$ of *variables*. A *literal* is a variable or the negation $\overline{x}$ of a variable $x$. In the following, $\overline{\overline{x}}$ is the same as $x$.

A *clause* is a finite set of literals, and a *formula* is a finite set of clauses. An *assignment* is a consistent set of literals.

Consider an assignment $\mu$ and formula $\varphi$.

A literal $l$ is *assigned by* $\mu$ if either $l$ or $\overline{l}$ is in $\mu$. We say that $\mu$

- is *total* if each variable in $P$ is assigned by $\mu$;
- *satisfies* a formula $\varphi$ if for each clause $C \in \varphi$, $C \cap \mu \neq \emptyset$.

A formula is *satisfiable* if there exists an assignment satisfying it.

Consider a partial order $\prec$ on the set of total assignments. Intuitively speaking, $\mu' \prec \mu$ means that $\mu'$ is preferred to $\mu$. Thus, for us, a *total assignment $\mu$ is optimal (with respect to $\prec$)* if

1. $\mu$ satisfies $\varphi$; and
2. there is no total assignment $\mu'$ satisfying $\varphi$ and with $\mu' \prec \mu$.

An *assignment $\mu$ is optimal (with respect to $\prec$)* if $\mu$ satisfies $\varphi$ and there exists a total and optimal assignment extending $\mu$.

## 3 OPT-DLL

As we anticipated in the introduction, OPT-DLL is like the standard DLL except for the heuristic. Given a formula $\varphi$, the basic idea of OPT-DLL is to first explore the search space where there can be a not yet ruled-out optimal assignment. In DLL, assuming that the current assignment is $\mu$ and that it is not the case that all the assignments extending $\mu$ are equally good, this amounts to knowing on which literal we have to branch.

To make these notions precise, consider a partially ordered set $S, \prec$ in which $S$ is a set of literals and $\prec$ is such that for each literal $l \in S$, either $l \prec \overline{l}$ or $\overline{l} \prec l$: If $\prec$ satisfies this condition, we say that $\prec$ is DLL-*compatible (with respect to $S$)*. For example, the partial order on $\{\overline{x}_0, x_0, \overline{x}_1, x_1\}$ such that

$$\overline{x}_0 \prec x_0, \overline{x}_1 \prec x_1, \overline{x}_1 \prec \overline{x}_0, \tag{1}$$

is DLL-compatible with respect to $\{x_0, \overline{x}_0, x_1, \overline{x}_1\}$. Notice that the condition that for each $l \in S$, either $l \prec \overline{l}$ or $\overline{l} \prec l$ ensures that both $l \in S$ and $\overline{l} \in S$. Given this, the set $S$ can be omitted from the specification of a DLL-compatible partially ordered set $S, \prec$.

The pseudo-code of OPT-DLL is represented in Figure 1, where:

- $\varphi$ is a formula; $\mu$ is an assignment; $\prec$ is a partial order DLL-compatible with respect to a set $S$ of literals;
- *assign*$(l, \varphi)$ returns the formula obtained from $\varphi$ by $(i)$ deleting the clauses $C \in \varphi$ with $l \in C$, and $(ii)$ deleting $\overline{l}$ from the other clauses in $\varphi$;
- *ChooseLiteral*$(\varphi, \mu, \prec)$ returns an unassigned literal $l$ such that
  - if there exists a variable in $S$ which is not assigned by $\mu$, then each literal $l'$ with $l' \prec l$ has to be assigned by $\mu$, and
  - is an arbitrary literal occurring in $\varphi$, otherwise.

**function** OPT-DLL($\varphi$,$\mu$,$\prec$)
1 **if** ($\emptyset \in \varphi$) **return** FALSE;
2 **if** ($\varphi = \emptyset$) **return** $\mu$;
3 **if** ($\{l\} \in \varphi$) **return** OPT-DLL(*assign*$(l, \varphi), \mu \cup \{l\}, \prec$);
4 $l$ := *ChooseLiteral*$(\varphi, \mu, \prec)$;
5 $v$ := OPT-DLL(*assign*$(l, \varphi), \mu \cup \{l\}, \prec$);
6 **if** ($v \neq$ FALSE) **return** $v$;
7 **return** OPT-DLL(*assign*$(\overline{l}, \varphi), \mu \cup \{\overline{l}\}, \prec$).

**Figure 1.** The algorithm of OPT-DLL.

OPT-DLL has to be invoked with $\varphi$ and $\mu$ set to the input formula and the empty set respectively. It is easy to see that if the set $S$ is empty, OPT-DLL is the same as DLL.

Assuming $\prec$ is (1), OPT-DLL checks the existence of an assignment satisfying $\varphi$ and extending one of $\{\overline{x}_1, \overline{x}_0\}$, $\{\overline{x}_1, x_0\}$, $\{x_1, \overline{x}_0\}$, $\{x_1, x_0\}$, following the order in which they are listed. Assuming $x_1, x_0$ are two variables encoding the values from 0 to 3, OPT-DLL will return

- an assignment with minimal corresponding value, if $\varphi$ is satisfiable, and
- FALSE otherwise.

Assuming $x_0, x_1$ represent the actions of going by car and by plane respectively, (1) encodes the fact that we prefer to not perform these actions, and that not going by plane is preferred to not going by car. Consequently, OPT-DLL will first look for an assignment where both actions are false, then one in which we use only the car, then one in which we use only the plane, and only finally for assignments where we have to use both the car and the plane.

As the example makes clear, the partial order on the set $S$ of literals induces a partial order on the set of total assignments. In the case of the example (1), if $\mu_0 = \{\overline{x}_0, \overline{x}_1\} \cup S_0$, $\mu_1 = \{\overline{x}_1, x_0\} \cup S_1$, $\mu_2 = \{x_1, \overline{x}_0\} \cup S_2$, $\mu_3 = \{x_1, x_0\} \cup S_3$ are four total assignments, we have

$$\mu_0 \prec \mu_1 \prec \mu_2 \prec \mu_3, \tag{2}$$

while, if $\mu$ and $\mu'$ are two total assignments assigning in the same way both $x_0$ and $x_1$, $\mu \not\prec \mu'$, i.e., $\mu$ and $\mu'$ are equally good.

Assuming $\prec$ is a given partial order on $S$, $\prec$ is extended to the set of total assignments as follows: If $\mu$ and $\mu'$ are total assignments, $\mu \prec \mu'$ if and only if

1. there exists a literal $l \in S$ with $l \in \mu$ and $\overline{l} \in \mu'$; and
2. for each literal $l' \in S \cap (\mu' \setminus \mu)$, there exists a literal $l \in S \cap (\mu \setminus \mu')$ such that $l \prec l'$.

The first condition says that two total assignments are not in partial order if they assign in the same way the literals in $S$. The second condition says that $\mu$ is preferred to $\mu'$ if for each literal $l' \in S$ with $l' \in \mu'$ and $\overline{l'} \in \mu$, $\mu$ contains a literal $l \in S$ with $l \in \mu$ and $\overline{l} \in \mu'$, and $l$ is preferred to $l'$ (i.e., $l \prec l'$).

In the case of (1), the above definition leads to the partial order on the set of total assignment satisfying (2).

We can now state the formal result that OPT-DLL returns an optimal assignment, assuming the input formula is satisfiable.

**Theorem 1** *Let $\varphi$ be a formula and $\prec$ a* DLL*-compatible partial order on a set of literals.* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns an optimal (with respect to $\prec$ extended to the set of total assignments) assignment if $\varphi$ is satisfiable, and returns* FALSE *otherwise.*

## 4 Solving optimization problems with OPT-DLL

Considering OPT-DLL and our definition of optimality given in Section 2, it is clear that OPT-DLL can solve only those optimization problems in which the partial order on the set of total assignments can be obtained as the extension of a DLL-compatible partial order on a set of literals. Indeed, this is not always possible: Assuming we have only two variables $x_0, x_1$, the total order on the set of total assignments $\{\overline{x}_1, \overline{x}_0\} \prec \{x_1, \overline{x}_0\} \prec \{x_1, x_0\} \prec \{\overline{x}_1, x_0\}$ is not obtainable as the result of the extension of a DLL-compatible partial order on a set of literals. Still, many important optimization problems can be easily modeled via a DLL-compatible partial order on a set of literals, and thus solved with OPT-DLL. Given a formula $\varphi$, we first consider MIN-ONE/MIN-ONE$_\subseteq$ and then MAX-SAT/MAX-SAT$_\subseteq$ problems: These problems are very interesting from an application perspective, as briefly described below. We also show how DISTANCE-SAT/DISTANCE-SAT$_\subseteq$ problems can be solved via OPT-DLL.

### 4.1 MIN-ONE and MIN-ONE$_\subseteq$

Let $S$ be a subset of the set $P$ of variables. Consider a satisfiable formula $\varphi$. Define MIN-ONE$^S(\varphi)$ (resp. MIN-ONE$^S_\subseteq(\varphi)$) to be the set of assignments $\mu$ satisfying $\varphi$ and having $\mu \cap S$ of minimal cardinality (resp. minimal). It is clear that MIN-ONE$^S(\varphi) \subseteq$ MIN-ONE$^S_\subseteq(\varphi)$.

In MIN-ONE (resp. MIN-ONE$_\subseteq$), the goal is to find an assignment $\mu$ in MIN-ONE$^P(\varphi)$ (resp. MIN-ONE$^P_\subseteq(\varphi)$). As pointed out in [8], MIN-ONE problems are interesting because various graph problems involving combinatorial optimization can be converted in linear time into them. In planning, if $\varphi$ is a formula encoding a planning as satisfiability problem, any assignment satisfying $\varphi$ corresponds to a sequence of (possibly parallel) actions achieving the goal: If $S$ is the set of action variables in $\varphi$,

1. if we want that as few as possible actions are executed, then we have to find an assignment in MIN-ONE$^S(\varphi)$;
2. if we want that no redundant sequence of (possibly parallel) actions is executed, then we have to find an assignment in MIN-ONE$^S_\subseteq(\varphi)$.

An assignment in MIN-ONE$^S_\subseteq(\varphi)$ can be easily computed via OPT-DLL, as stated by the following theorem, consequence of Theorem 1.

**Theorem 2** *Let $S$ be a subset of $P$, and let $\varphi$ be a formula. Let $\prec$ be the* DLL*-compatible partial order such that $l \prec l'$ iff $l = \overline{x}$, $l' = x$, $x \in S$. If* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns an assignment $\mu$ then $\mu \in$* MIN-ONE$^S_\subseteq(\varphi)$. *If* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns* FALSE *then $\varphi$ is unsatisfiable.*

Intuitively speaking, DLL is forced to split first on the variables in $S$, assigning them to false.

An assignment in MIN-ONE$^S(\varphi)$ can be computed via OPT-DLL, assuming we have a formula encoding the objective function. This amounts to define a formula *adder*$(S)$ such that[3]

1. the only variables in common to *adder*$(S)$ and $\varphi$ are those in $S$;
2. if $n = \lceil log_2(|S| + 1) \rceil$, *adder*$(S)$ contains $n$ new variables $b_{n-1}, \ldots, b_0$; and
3. for any total assignment $\mu$ to the variables in $\varphi$, there exists a unique total assignment $\nu$ to the variables in *adder*$(S)$ such that
   (a) $\nu$ satisfies *adder*$(S)$;
   (b) $\mu$ and $\nu$ assign in the same way the variables in $S$, i.e., $\mu \cap S = \nu \cap S$;
   (c) $|\nu \cap S| = \sum_{i=0}^{n-1} \nu(b_i) \times 2^i$, where $\nu(b_i)$ is 1 if $\nu$ assigns $b_i$ to true, and is 0 otherwise.

*adder*$(S)$ can be realized in polynomial time in many ways, see, e.g., [17]. If the above conditions are satisfied, we say that *adder*$(S)$ is an *adder of $S$ with output $b_{n-1}, \ldots, b_0$*.

**Theorem 3** *Let $S$ be a subset of $P$. Let adder$(S)$ be an adder of $S$ with output $b_{n-1}, \ldots, b_0$. Let $\varphi$ be a formula. Let $\prec$ be the* DLL*-compatible partial order on $\{b_{n-1}, \ldots, b_0\}$ such that for each $i \in [0, n-1]$, $\overline{b}_i \prec b_i$, and, if $i \neq 0$, $\overline{b}_i \prec \overline{b}_{i-1}$. If* OPT-DLL$(\varphi \cup$ *adder*$(S), \emptyset, \prec)$ *returns an assignment $\mu$ then $\mu \cap P \in$* MIN-ONE$^S(\varphi)$. *If* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns* FALSE *then $\varphi$ is unsatisfiable.*

Intuitively speaking, assuming no literal in $\{b_{n-1}, \ldots, b_0\}$ is assigned as unit, OPT-DLL first explores the branches with the variables $b_{n-1}, \ldots, b_0$ assigned to false; If all such branches fail, then OPT-DLL explores the branches in which $b_{n-1}, \ldots, b_1$ are assigned to false while $b_0$ is assigned to true; If also these branches fail, then OPT-DLL explores the branches in which $b_{n-1}, \ldots, b_2, b_0$ are assigned to false while $b_1$ is assigned to true; and so on and so forth.

### 4.2 MAX-SAT and MAX-SAT$_\subseteq$

Consider a formula $\varphi$. Let $S$ be a subset of the clauses in $\varphi$. Intuitively speaking, we consider the problem of satisfying $S$ and "as many as possible" clauses in $\varphi \setminus S$. Formally, define MAX-SAT$^S(\varphi)$ (resp. MAX-SAT$^S_\subseteq(\varphi)$) to be the set of assignments $\mu$ satisfying each clause in $S$ and such that the set $\{C : C \in \varphi \setminus S, C \cap \mu \neq \emptyset\}$ is of maximal cardinality (resp. maximal). It is clear that MAX-SAT$^S(\varphi) \subseteq$ MAX-SAT$^S_\subseteq(\varphi)$.

In MAX-SAT (resp. MAX-SAT$_\subseteq$), the goal is to find an assignment $\mu$ in MAX-SAT$^\emptyset(\varphi)$ (resp. MAX-SAT$^\emptyset_\subseteq(\varphi)$). The problem of determining an assignment in MAX-SAT$^S(\varphi)$/MAX-SAT$^S_\subseteq(\varphi)$ can be easily reduced to a MIN-ONE/MIN-ONE$_\subseteq$ problem by considering the formula $x(\varphi, S)$, obtained from $\varphi$ by adding a newly created variable $v_i$ to the $i$-th clause in $\varphi \setminus S$.

For example, if $\varphi = \{\{\overline{x}_0, \overline{x}_1\}, \{\overline{x}_0, x_1\}, \{x_0\}\}$ and $S = \{\{x_0\}\}$, $x(\varphi, S)$ is $\{\{v_1, \overline{x}_0, \overline{x}_1\}, \{v_2, \overline{x}_0, x_1\}, \{x_0\}\}$

**Theorem 4** *Let $\varphi$ be a formula. Let $S$ be a subset of $\varphi$. Let $V$ be the set of variables in $x(\varphi, S)$ and not in $\varphi$. The following equalities hold:*

$$\text{MAX-SAT}^S(\varphi) = \{\mu \cap P : \mu \in \text{MIN-ONE}^V(x(\varphi, S))\},$$

*and*

$$\text{MAX-SAT}^S_\subseteq(\varphi) = \{\mu \cap P : \mu \in \text{MIN-ONE}^V_\subseteq(x(\varphi, S))\}.$$

MAX-SAT is arguably the most studied problem in the SAT literature after the SAT problem itself. MAX-SAT$_\subseteq$ problems arise in many settings. For instance, in formal verification, if $\varphi$ is a formula encoding an initial specification of a system, and if $\varphi'$ encodes a refinement of the initial specification, a standard verification task is to prove that the refinement $\varphi'$ is compatible with $\varphi$, i.e., that $\varphi \cup \varphi'$ is satisfiable. If $\varphi \cup \varphi'$ is unsatisfiable, one goal is to find "as large as possible" parts of the refinement which are consistent with the initial design: Such parts correspond to the assignments in MAX-SAT$^\varphi_\subseteq(\varphi \cup \varphi')$.

[3] The specification of *adder*$(S)$ will be used also in Section 4.3, where $S$ is assumed to be an assignment.

### 4.3 DISTANCE-SAT and DISTANCE-SAT$_\subseteq$

DISTANCE-SAT [3] is another optimization problem in which, given a formula $\varphi$ and an assignment $\mu$, the goal is to find an assignment $\mu'$ satisfying $\varphi$ and differing "as little as possible" from $\mu$. Here we consider also its variant DISTANCE-SAT$_\subseteq$. Formally: Let $\mu$ be an assignment. Define DISTANCE-SAT$(\varphi, \mu)$ (resp. DISTANCE-SAT$_\subseteq(\varphi, \mu)$) to be the set of assignments $\mu'$ satisfying $\varphi$ and having the set $\{l : l \in \mu, \bar{l} \in \mu'\}$ of minimal cardinality (resp. minimal). It is clear that DISTANCE-SAT$^S(\varphi) \subseteq$ DISTANCE-SAT$^S_\subseteq(\varphi)$.

**Theorem 5** *Let $\varphi$ be a formula. Let $\mu$ be an assignment. The following two facts hold:*

1. *Let $\prec$ be the* DLL*-compatible partial order such that $l \prec \bar{l}$ if $l \in \mu$. If* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns an assignment $\mu'$ then $\mu' \in$* DISTANCE-SAT$_\subseteq(\varphi, \mu)$. *If* OPT-DLL$(\varphi, \emptyset, \prec)$ *returns* FALSE *then $\varphi$ is unsatisfiable.*
2. *Let adder$(\mu)$ be an adder of $\mu$ with output $b_{n-1}, \ldots, b_0$. Let $\prec$ be the* DLL*-compatible partial order on $\{b_{n-1}, \ldots, b_0\}$ such that for each $i \in [0, n-1]$, $b_i \prec \bar{b}_i$, and, if $i \neq 0$, $b_i \prec b_{i-1}$. If* OPT-DLL$(\varphi \cup$ *adder*$(\mu), \emptyset, \prec)$ *returns an assignment $\mu'$ then $\mu' \cap P \in$* DISTANCE-SAT$(\varphi, \mu)$. *If* OPT-DLL$(\varphi \cup$ *adder*$(\mu), \emptyset, \prec)$ *returns* FALSE *then $\varphi$ is unsatisfiable.*

## 5 Implementation and experimental results

| | instance | $\#C$ | OPBDP | PBS4 | MSAT+ | optsat | $\#C_\subseteq$ | optsat |
|---|---|---|---|---|---|---|---|---|
| 1 | bcomp5 | 39 | 0.95 | 8.87 | 0.4 | 7.08 | 85 | 0 |
| 2 | bmax6 | 61 | 120.05 | TIME | 8.42 | 274.13 | 131 | 0 |
| 3 | ibm2 | 940 | TIME | TIME | 19.73 | TIME | 1054 | 0.03 |
| 4 | ibm3 | 6356 | TIME | – | TIME | 79.17 | 6422 | 0.65 |
| 5 | gal8 | | MEM | – | SF | TIME | 14207 | 2.87 |
| 6 | 3blocks | 56 | 282.03 | 0.19 | 0.29 | 2.2 | 58 | 0.06 |
| 7 | 4blocksb | 66 | TIME | 0.61 | 0.24 | 5.81 | 66 | 0.09 |
| 8 | 4blocks | 108 | TIME | 115.96 | 50.94 | TIME | 116 | 0.39 |
| 9 | large.c | 265 | TIME | 0.94 | 0.96 | 1.5 | 272 | 0.3 |
| 10 | large.d | 431 | TIME | – | 7.71 | 99.75 | 443 | 1.1 |
| 11 | log.a | 135 | TIME | TIME | 1.39 | 7.53 | 135 | 0.04 |
| 12 | log.b | 138 | TIME | TIME | 8.99 | TIME | 138 | 0.04 |
| 13 | rock.a | 65 | TIME | 1.21 | 0.2 | 9.39 | 65 | 0.01 |
| 14 | rock.b | 69 | TIME | 0.14 | 0.27 | 5.9 | 69 | 0.01 |
| 15 | r2b3.1 | 141 | 32.76 | 0.04 | 0.2 | 0.17 | 141 | 0.04 |
| 16 | r2b3.2 | 138 | 67.14 | 0.03 | 0.08 | 0.14 | 138 | 0.03 |
| 17 | r3b1.1 | 119 | TIME | 8.57 | 1.3 | 6.73 | 119 | 1.29 |
| 18 | r3b1.2 | 126 | TIME | 7.09 | 0.82 | 8.49 | 126 | 0.21 |
| 19 | r3b2.1 | 217 | TIME | 0.33 | 0.46 | 0.71 | 217 | 0.09 |
| 20 | r3b2.2 | 206 | TIME | 0.25 | 0.53 | 0.73 | 206 | 0.08 |
| 21 | qg1-8 | 64 | TIME | 81.46 | 31.06 | 85.67 | 64 | 6.87 |
| 22 | qg2-7 | 49 | 75.03 | 0.23 | 0.27 | 0.72 | 49 | 0.16 |
| 23 | qg2-8 | 64 | MEM | 54.26 | 21.83 | 29.56 | 64 | 20.83 |
| 24 | qg3-8 | 64 | 19.62 | 0.24 | 0.1 | 0.61 | 64 | 0.02 |
| 25 | qg4-9 | 81 | TIME | 53.12 | 19.36 | 250.69 | 81 | 0.2 |
| 26 | qg5-11 | 121 | MEM | 0.25 | 0.43 | 0.86 | 121 | 0.15 |

**Table 1.** MIN-ONE (columns 3-7) and MIN-ONE$_\subseteq$ (columns 8-9) problems.

The implementation of a solver based on our ideas requires the modification of the heuristic of a DLL based SAT solver. In our case, we selected ZCHAFF [13], the 2004 version. Such choice is motivated by our interest in solving large problems coming from applications, and by the fact that we already had some experience in hacking ZCHAFF code.

| | instance | $\#C$ | BF | OPBDP | PBS4 | MSAT+ | optsat | $\#C_\subseteq$ | optsat |
|---|---|---|---|---|---|---|---|---|---|
| 1 | barrel3 | 941 | 0.23 | 2.04 | 0.88 | 0.12 | 0.9 | 941 | 0.09 |
| 2 | barrel4 | 2034 | 0.65 | 47.59 | 11.67 | 0.34 | 21.19 | 2034 | 1.03 |
| 3 | barrel5 | 5382 | 21.42 | MEM | MEM | 24.01 | 177.11 | 5382 | 122.76 |
| 4 | barrel6 | 8930 | 213.60 | MEM | – | 95.56 | 896.45 | 8930 | 1438.08 |
| 5 | barrel7 | 13764 | SF | MEM | – | 285.55 | 435.46 | | TIME |
| 6 | lmult0 | 1205 | 0.39 | 13.05 | 1.45 | 0.16 | 7.35 | 1205 | 0 |
| 7 | lmult2 | 3524 | 57.11 | TIME | TIME | 6.7 | 16.46 | 3524 | 0.1 |
| 8 | lmult4 | 6068 | 261.74 | MEM | – | 35.34 | 98.05 | 6068 | 27.56 |
| 9 | lmult6 | 8852 | 774.08 | MEM | – | 157.02 | 609.07 | | TIME |
| 10 | lmult8 | 11876 | SF | MEM | – | 297.32 | 704.08 | | TIME |
| 11 | qvar8 | 2272 | 0.62 | MEM | 17.67 | 2.95 | 36 | 2272 | 3.58 |
| 12 | qvar10 | 5621 | 2.21 | MEM | 234.97 | 55.54 | 156.44 | 5621 | 48.25 |
| 13 | qvar12 | 7334 | 6.2 | MEM | – | 36.8 | 74.49 | 7334 | 238.36 |
| 14 | qvar14 | 9312 | SF | MEM | – | 117.25 | 815.66 | 9312 | 942.48 |
| 15 | qvar16 | 6495 | SF | MEM | – | 51.33 | 117.31 | 6495 | 261.83 |
| 16 | c432 | 1114 | 131.06 | TIME | 7.22 | 0.24 | 7.6 | 1114 | 0.74 |
| 17 | c499 | 1869 | TIME | TIME | 100.41 | 0.8 | 4.59 | 1869 | 4.3 |
| 18 | c880 | 2589 | TIME | TIME | 320.96 | 5.54 | 38.91 | 2589 | 16.72 |
| 19 | c1355 | 3661 | TIME | TIME | TIME | 80.09 | 21.2 | 3661 | 30.86 |
| 20 | c1908 | 5095 | TIME | MEM | TIME | 58.01 | 165.99 | 5095 | 129.95 |
| 21 | c2670 | 6755 | TIME | MEM | – | 63.64 | 100.31 | 6755 | 359.52 |
| 22 | c3540 | 9325 | TIME | MEM | – | 242.02 | 786.33 | | TIME |
| 23 | u-bw.a | 3290 | 7.81 | TIME | 249 | 209.03 | 178.18 | 3288 | 0.04 |
| 24 | u-bw.b | | TIME | MEM | – | TIME | TIME | 11487 | 87.61 |
| 25 | u-log.a | 5783 | TIME | MEM | TIME | 59.65 | 179.3 | 5782 | 1.35 |
| 26 | u-log.b | 6428 | TIME | MEM | – | 35.37 | 144.83 | 6428 | 19.68 |
| 27 | u-log.c | 9506 | TIME | MEM | – | 383.65 | 731.87 | 9506 | 76.89 |
| 28 | u-rock.a | 1691 | 13.29 | TIME | 41.29 | 206.56 | 6.26 | 1690 | 4.79 |

**Table 2.** MAX-SAT and MAX-SAT$_\subseteq$ problems, columns 3-8 and 9-10 resp.

For MIN-ONE$_\subseteq$ problems, the heuristics VSIDS of ZCHAFF has been modified by simply selecting the unassigned literal $l$ with highest VSIDS score, and then assigning the variable $x$ in $l$ to false. For MAX-SAT$_\subseteq$, if there exists an unassigned literal $l$ in $x(\varphi, S)$ and not in $\varphi$, the one with the highest VSIDS score is selected and the variable in it is assigned to false. Otherwise, the unassigned literal $l$ with highest VSIDS score is selected and assigned to true. Analogous modifications have been done on VSIDS in order to solve MIN-ONE/MAX-SAT problems.

The solution of MIN-ONE/MAX-SAT problems also required the implementation of a function *adder*$(S)$ as specified in Section 4.1. As we already said, there are various ways to implement such function. We used the method described in [17] which takes linear time in the size of $S$. We call optsat the resulting system.[4] Beside the modification in the heuristic, we had also to modify ZCHAFF preprocessor in order to disable the assignment of pure literals.

About the other solvers, we initially considered both dedicated solvers for MAX-SAT problems —like BF [7], MAXSOLVER [18], WCSP [14]— and generic Pseudo-Boolean solvers —like OPBDP ver. 1.1.1 [4], PBS ver. 2.1 and ver. 4 [1], MSAT+ (abbreviation of MINISAT+) based on MINISAT ver. 1.13 [10]. MSAT+ was the solver able to prove unsatisfiability and optimality to a larger number of instances than all the other solvers that entered into the last Pseudo-Boolean evaluation [12]. Considering the dedicated solvers for MAX-SAT, we discarded MAXSOLVER and WCSP after an initial analysis because they seem to be tailored for relatively small typically randomly generated problems, and are thus not suited to deal with problems coming from applications. About the Pseudo-Boolean solvers, we do not show the results for PBS ver 2.1 because it is almost always slower than ver 4.0.

Each solver has been run using its default settings. All the ex-

[4] Available at `http://www.star.dist.unige.it/~marco/OPTSAT/`.

periments have been run on a Linux box equipped with a Pentium IV 2.4GHz processor and 1GB of RAM. The results for MIN-ONE/MIN-ONE$_\subseteq$ and MAX-SAT/MAX-SAT$_\subseteq$ problems are reported in Tables 1 and 2 respectively.[5] CPU time is measured in seconds; timeout has been set to 1800 seconds. In the tables, "TIME" indicates that the solver does not solve the instance within the time limit; "MEM" indicates that the solver exceeds all the available memory; "SF" indicates that the solver exits abnormally; "–" indicates that the solver returns an incorrect result.

Considering the results for MIN-ONE problems in columns 4-7 of Table 1, we see that our solver optsat performs much better than all the other solvers except for MSAT+. OPBDP solves a few instances, PBS times out or outputs an incorrect result on large instances, our solver times out on four instances, while MSAT+ times out on 1 instance and on another instance it exits abnormally.

Considering MIN-ONE$_\subseteq$ problems, the results of our solver are shown in the last column of the table. Given that for any formula $\varphi$ in the table $\emptyset \neq$ MIN-ONE$(\varphi) \subseteq$ MIN-ONE$_\subseteq(\varphi)$ and that it is not possible to codify MIN-ONE$_\subseteq$ problems in the other solvers we considered, it makes sense to compare the performances of our solver with the performances of the other solvers in columns 4-6. The first observation is that optsat is much faster than all the other systems: Almost all the problems are solved in less than 1s. Two other observations are in order:

1. Comparing optsat results in columns 7 and 9 we see that our solver is much faster in solving MIN-ONE$_\subseteq$ than MIN-ONE problems. This could have been expected given that handling MIN-ONE problems requires the encoding of adders counting the number of variables set to true, and many of the examples have more than a thousand variables (the "gal8" instance has $\geq 58000$ variables).
2. comparing the cardinality $\#C_\subseteq$ in column 8 (resp. $\#C$ in column 3) of the optimal assignment returned by optsat when solving a MIN-ONE$_\subseteq$ (resp. MIN-ONE) problem, we see that for most instances $\#C = \#C_\subseteq$.

Considering the results for MAX-SAT problems in Table 2, we see that our solver optsat performs much better than all the other solvers, including the dedicated ones, except for MSAT+. In comparison to MSAT+, our solver is slower of a factor on most instances, but is also faster on some instances. As in the previous case, the performances of optsat on the same instances treated as MAX-SAT$_\subseteq$ problems are shown in the last column. Differently from the previous case, optsat is slower in solving MAX-SAT$_\subseteq$ than MAX-SAT problems except for the planning instances (rows (23-28)). We do not yet have a clear understanding of why this happens. We believe that this is related to the fact that for all the instances that we considered, $\#C/\#C_\subseteq$ (representing the cardinality of the set returned by optsat when solving a MAX-SAT/MAX-SAT$_\subseteq$ problem) are very close to number of clauses in the original SAT instance, but this is still subject of investigation.

## 6 Conclusions and future work

In this paper we showed that DLL can be used to solve optimization problems by simply imposing an ordering on the literals to be used while branching. We specifically considered MIN-ONE/MIN-ONE$_\subseteq$/MAX-SAT/MAX-SAT$_\subseteq$/DISTANCE-SAT/DISTANCE-SAT$_\subseteq$ problems, but it is clear that any optimization problem where the ordering on the set of total assignments can be obtained by extending a partial order on a set of literals can be handled by OPT-DLL. In particular, all the problems where the optimality condition is expressed via an objective function $f$, can be handled by OPT-DLL, provided we have a formula encoding the value of $f$. This is indeed the case, e.g., for WEIGHTED-MAX-SAT where the encoding is illustrated, e.g., in [17].

We implemented our ideas using ZCHAFF as engine, and the encoding of [17] to solve MIN-ONE/MAX-SAT problems. The results are positive and encouraging. We believe that even better performances will be obtained by using MINISAT and, for MIN-ONE/MAX-SAT problems, using the encoding presented in [2, 15] of the objective function. Some of these encoding produce formulas of a bigger size, but they should lead to better performances of the back-end solver: See [2, 15] for more details.

[5] In Table 1 (1-5) are Formal Verification instances ((1-2) from the Bejing'96 competition, (3-5) by Ofer Shtrichman); (6-14) are planning problems from SATPLAN; (15-20) are Data Encryption Standard problems; (21-26) are quasi group. In Table 2, (1-13) are Bounded Model Checking (BMC) problems used in the original BMC paper; (16-22) are miter-based circuit equivalence benchmarks by Joao Marques-Silva; (23-28) are planning problems from SATPLAN.

## ACKNOWLEDGEMENTS

This work is partially supported by MIUR.

## REFERENCES

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, 'PBS: A backtrack search pseudo-Boolean solver', in *Proc. SAT*, (2002).
[2] Olivier Bailleux and Yacine Boufkhad, 'Efficient CNF encoding of Boolean cardinality constraints.', in *Proc. CP*, pp. 108–122, (2003).
[3] Olivier Bailleux and Pierre Marquis, 'Some Computational Aspects of DISTANCE-SAT', *Journal of Automated Reasoning*, to appear, (2006).
[4] P. Barth, 'A Davis-Putnam enumeration algorithm for linear pseudo-boolean optimization', Technical report, Max Plank Instutute for Computer Scince, (1995). technical Report MPI-I-95-2-2003.
[5] D. Le Berre and L. Simon, 'Fifty-five solvers in Vancouver: The SAT 2004 competition', in *Proc. SAT (Selected Papers)*, pp. 321–344, (2004).
[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, 'Symbolic model checking without BDDs', in *Proc. TACAS*, (1999).
[7] Brian Borchers and Judith Furman, 'A two-phase exact algorithm for max-SAT and weighted max-SAT problems.', *J. Comb. Optim.*, **2**(4), 299–306, (1998).
[8] N. Creignou, S. Khanna, and M. Sudan, 'Complexity classifications of Boolean constraint satisfaction problems', *SIAM*, (2001).
[9] M. Davis, G. Logemann, and D. Loveland, 'A machine program for theorem proving', *Journal of the ACM*, **5**(7), (1962).
[10] Niklas Eén and Niklas Sörensson, 'Translating pseudo-Boolean constraints into SAT', *Journal on Satisfiability, Boolean Modeling and Computation*, (2006).
[11] Henry Kautz and Bart Selman, 'Planning as satisfiability', in *Proc. ECAI*, pp. 359–363, (1992).
[12] Vasco Miguel Manquinho and Olivier Roussel, 'The first evaluation of pseudo-Boolean solvers (PB05)', *Journal on Satisfiability, Boolean Modeling and Computation*, (2006).
[13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, 'Chaff: Engineering an Efficient SAT Solver', in *Proc. DAC*, (2001).
[14] P. Meseguer S. D. Givry, J. Larrosa and T. Schieux, 'Solving Max-SAT as weighted CSP', pp. 363–376, (2003).
[15] Carsten Sinz, 'Towards an optimal cnf encoding of Boolean cardinality constraints.', in *Proc. CP*, pp. 827–831, (2005).
[16] John K. Slaney and Toby Walsh, 'Phase transition behavior: from decision to optimization', in *Proc. SAT*, (2002).
[17] Joost P. Warners, 'A linear-time transformation of linear inequalities into conjunctive normal form.', *Inf. Process. Lett.*, **68**(2), 63–69, (1998).
[18] Z. Xing and W. Zhang, 'MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability', *Artificial Intelligence*, **164**(1-2), 47–80, (2005).