Journal of Artificial Intelligence Research 31 (2008) 1-32 Submitted 5/07; published 01/08

MINIMAXSAT: An Efficient Weighted Max-SAT Solver

Federico Heras FHERAS@LSI.UPC.EDU
Javier Larrosa LARROSA@LSI.UPC.EDU
Albert Oliveras OLIVERAS@LSI.UPC.EDU
Technical University of Catalonia, LSI Department
Jordi Girona 1-3, 08034, Barcelona, Spain.

Abstract

In this paper we introduce MINIMAXSAT, a new Max-SAT solver that is built on top of MINISAT+. It incorporates the best current SAT and Max-SAT techniques. It can handle hard clauses (clauses of mandatory satisfaction as in SAT), soft clauses (clauses whose falsification is penalized by a cost as in Max-SAT) as well as pseudo-boolean objective functions and constraints. Its main features are: learning and backjumping on hard clauses; resolution-based and substraction-based lower bounding; and lazy propagation with the two-watched literal scheme. Our empirical evaluation comparing a wide set of solving alternatives on a broad set of optimization benchmarks indicates that the performance of MINIMAXSAT is usually close to the best specialized alternative and, in some cases, even better.

1. Introduction

Max-SAT is the optimization version of SAT where the goal is to satisfy the maximum number of clauses. It is considered one of the fundamental combinatorial optimization problems and many important problems can be naturally expressed as Max-SAT. They include academic problems such as *max cut* or *max clique*, as well as real problems in domains like *routing*, *bioinformatics*, *scheduling* or *electronic markets*.

There is a long tradition of theoretical work about the structural complexity (Papadimitriou, 1994) and approximability (Karloff & Zwick, 1997) of Max-SAT. Most of this work is restricted to the simplest case in which all clauses are equally important (*i.e.*, unweighted Max-SAT) and have a fixed size (mainly binary or ternary). From a practical point of view, significant progress has been made in the last 3 years (Shen & Zhang, 2004; Larrosa & Heras, 2005; Larrosa, Heras, & de Givry, 2007; Xing & Zhang, 2005; Li, Manyà, & Planes, 2005, 2006). As a result, there is a handful of new solvers that can deal, for the first time, with instances involving hundreds of variables.

The main motivation of our work comes from the study of Max-SAT instances modelling real-world problems. We usually encounter three features:

- The satisfaction of all clauses does not have the same importance, so each clause needs to be associated with a weight that represents the cost of its violation. In the extreme case, which often happens in practice as observed by Cha, Iwama, Kambayashi, and Miyazaki (1997), there are clauses whose satisfaction is mandatory. They are usually modelled by associating a very high weight with them.

- Literals do not appear randomly along the clauses. On the contrary, it is easy to identify patterns, symmetries or other kinds of structures.

- In some problems there are mandatory clauses that reduce dramatically the number of feasible assignments, so the optimization part of the problem only plays a secondary role. However, in some other problems mandatory clauses are trivially satisfiable and the real difficulty lays on the optimization part.

When we look at current Max-SAT solvers, we find that none of them is robust over these three features. For instance, Li et al.'s (2005, 2006) solvers are restricted to formulas in which all clauses are equally important (i.e. unweighted Max-SAT), Shen and Zhang's (2004) one is restricted to binary clauses, the one described by Larrosa et al. (2007) seems to be efficient on very overconstrained problems (*i.e.*, only a small fraction of the clauses can be simultaneously satisfied), while the one by Alsinet, Manyà, and Planes (2005) seems to be efficient on slightly overconstrained problems (*i.e.* almost all the clauses can be satisfied). The solver described by Argelich and Manya (2007), developed in parallel to the research described in this paper, can handle mandatory clauses and is the only one that incorporates some learning, so it seems to perform well on structured problems. However, all non-mandatory clauses must have the same weight. Finally, approaches based on translating a Max-SAT instance into a SAT instance and solve them with a SAT solver seem to be effective in highly structured problems in which almost all clauses are mandatory (Fu & Malik, 2006; Le Berre, 2006).

In this paper we introduce MINIMAXSAT, a new weighted Max-SAT solver that incorporates the current best SAT and Max-SAT techniques. It is build on top of MINISAT+ (Eén & Sörensson, 2006), so it borrows its capability to deal with pseudo-boolean problems and all the MINISAT (Eén & Sörensson, 2003) features processing mandatory clauses such as learning and backjumping. We have extended it allowing it to deal with weighted clauses, while preserving the two-watched literal lazy propagation method. The main original contribution of MINIMAXSAT is that it implements a novel and very efficient lower bounding technique. Specifically, it applies unit propagation in order to detect disjoint subsets of mutually inconsistent clauses as done by Li et al. (2006). Then it simplifies the problem following Larrosa and Heras (2005), Heras and Larrosa (2006), Larrosa et al. (2007) in order to increment the lower bound. However, while in those works only the clauses that accomplish specific patterns are transformed, in MINIMAXSAT there is no need to define such patterns.

The structure of the paper is as follows: Section 2 provides preliminary definitions on SAT and Section 3 presents state-of-the-art solving techniques incorporated in a modern SAT solver such as MINISAT. Then, Section 4 presents preliminary definitions on Max-SAT and Section 5 overviews MINIMAXSAT. After that, Sections 6 and 7 focus on its lower bounding and additional features, respectively. In Section 8 we present the benchmarks used in our empirical evaluation and we report the experimental results. Finally, Section 9 presents related work and Section 10 concludes and points out possible future work.

2. Preliminaries on SAT

In the sequel $X = \{x_1, x_2, \ldots, x_n\}$ is the set of boolean variables. A *literal* is either a variable x_i or its negation $\bar{x}_i$. The variable to which literal l refers is noted $var(l)$. Given a literal l, its negation $\bar{l}$ is $\bar{x}_i$ if l is x_i and is x_i if l is $\bar{x}_i$. A *clause* C is a disjunction of literals. The *size* of a clause, noted $|C|$, is the number of literals that it has. The set of variables that appear in C is noted $var(C)$. Sometimes we associate a subscript Greek letter to a clause (e.g. $(x_i \vee x_j)_\alpha$) in order to facilitate future references of such clause.

Algorithm 1: DPLL basic structure.

```
Function Search() : boolean
    InitQueue( ) ;
    Loop
        UP( ) ;
        if Conflict then
            AnalyzeConflict( ) ;
            if Top Conflict then return false ;
            else
                LearnClause( ) ;
                Backjump( ) ;
        else if all variables assigned then return true ;
        else
            l := SelectLiteral( ) ;
            Enqueue(Q, l) ;
```

An *assignment* is a set of literals not containing a variable and its negation. Assignments of maximal size n are called *complete*, otherwise they are called *partial*. Given an assignment $\mathcal{A}$, a variable x is *unassigned* if neither x nor $\bar{x}$ belong to $\mathcal{A}$. Similarly, a literal l is *unassigned* if $var(l)$ is unassigned.

An assignment *satisfies* a literal iff it belongs to the assignment, it *satisfies* a clause iff it satisfies one or more of its literals and it *falsifies* a clause iff it contains the negation of all its literals. In the latter case we say that the clause is *conflicting* as it always happens with the empty clause, noted $\square$. A boolean formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a set of clauses representing their conjunction. A model of $\mathcal{F}$ is a complete assignment that satisfies all the clauses in $\mathcal{F}$.

If $\mathcal{F}$ has a model, we call it *satisfiable*, otherwise we say it is *unsatisfiable*. Moreover, if all complete assignments satisfy $\mathcal{F}$, we say that $\mathcal{F}$ is a *tautology*.

Clauses of size one are called *unit clauses* or simply *units*. When a formula contains a unit l, it can be simplified by removing all clauses containing l and removing $\bar{l}$ from all the clauses where it appears. The application of this rule until quiescence is called *unit propagation* (UP) and it is well recognized as a fundamental propagation technique in all current SAT solvers.

Another well-known rule is *resolution*, which, given a formula containing two clauses of the form $(x \vee A), (\bar{x} \vee B)$ (called *clashing clauses*), allows one to add a new clause $(A \vee B)$ (called *the resolvent*).

3. Overview of State-of-the-art DPLL-based SAT Solvers

In this section we overview the architecture of SAT solvers based on the DPLL (Davis, Logemann, & Loveland, 1962) procedure. This procedure, currently regarded as the most efficient complete search procedure for SAT, performs a systematic depth-first search on the space of assignments. An internal node is associated to a partial assignment and its two successors are obtained by selecting an unassigned variable x and extending the current assignment with x and $\bar{x}$, respectively. At each visited node, new units are derived due to the application of unit propagation (UP). If that leads

Algorithm 2: Unit Propagation.

Function *UP(Q) : Conflict*
  while (*Q contains non-propagated literals*) **do**
13    $l :=$ GetFirstNonPropagatedLit(Q); MarkAsPropagated(l) ;
14   **foreach** *clause* $C \vee \bar{l}$ *that becomes unit or falsified* **do**
15   **if** $C \vee \bar{l}$ *becomes a unit* q **then** Enqueue(Q, q) ;
16   **else if** $C \vee \bar{l}$ *becomes falsified* **then return** *Conflict* ;
  return *None* ;

to a conflicting clause, the procedure backtracks, performing non-chronological backtracking and clause learning, as originally proposed by Silva and Sakallah (1996).

An algorithmic description of the DPLL procedure appears in Algorithm 1. The algorithm uses a propagation queue Q which contains all units pending propagation and also contains a representation of the current assignment.

First, propagation queue Q is filled with the units contained in the original formula (line 1). The main loop starts in line 2 and at each iteration procedure UP is in charge of propagating all pending units (line 3). If a conflicting clause is found (line 4), the conflict is analyzed (line 5) and as a result a new clause is *learned* (i.e, inferred and recorded, line 7).

Then, the procedure backtracks, using the propagation queue Q to undo the assignment until exactly one of the literals of the learned clause becomes unassigned (line 8). If one can further backtrack while still maintaining this condition, it is advantageous to do so (this is commonly referred to as *backjumping* or *non-chronological backtracking*, see Silva & Sakallah, 1996). If UP leads to no conflict, a new unassigned literal is selected to extend the current partial assignment. The new literal is added to Q (line 10) and a new iteration takes place.

The procedure stops when a complete assignment is found (line 9) or when a top level conflict is found (line 6). In the first case, the procedure returns $true$ which indicates that a model has been found, while in the second case it returns $false$ which means that no model exists for the input formula.

The performance of DPLL-based SAT solvers was greatly improved in 2001, when the SAT solver CHAFF (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001) incorporated the *two-watched literal scheme* for efficient unit propagation, the *First UIP scheme* (Zhang, Madigan, Moskewicz, & Malik, 2001) for clause learning and the cheap *VSIDS* branching heuristic. Currently, most state-of-the-art SAT solvers, like MINISAT (Eén & Sörensson, 2003), implement small variations of all these three features. In the following we describe them in more depth.

3.1 Unit Propagation

The aim of unit propagation is twofold: on the one hand, it finds all clauses that have become units due to the current assignment, and on the other hand, it detects whether some clause has become conflicting. A more concrete procedure is given in Algorithm 2. While non-propagated literals exist in Q, it picks the oldest one l and marks it as propagated (line 13). Then all clauses containing $\bar{l}$ that may have become falsified or units are traversed (we will later describe how these clauses are detected). If one of such clauses becomes a unit q, it is enqueued in Q to be propagated later (line

15). The procedure iterates until there are no more units to propagate or until a conflicting clause is found (line 16).

There are two types of literals in Q: *decision literals* are those that the algorithm has heuristically selected and assigned at a branching point (lines 11 and 12 in Algorithm 1); *consequence literals* are those which are added because they are logical consequences of previous decision literals (line 15). MINISAT uses a non-standard queue to handle units pending propagation. Unlike classical queues, after fetching an element, it is not removed, but just marked as such. Consequently, Q is formed by two sets of elements: the already propagated literals and the literals pending propagation. The advantage of such strategy is that at any execution point, Q also contains the current assignment. Besides, the propagated literals of Q are divided into *decision levels*. Each decision level contains a decision literal and the set of its related consequences. Furthermore, a literal l is associated with the original clause that caused its propagation and it is noted as $l(\alpha)$; such a clause is usually referred to as the *reason* of l. Note that a decision literal l does not have a *reason* and will be represented as l^d.

Example 1 *Consider the formula* $\{(\bar{x}_1 \vee x_2)_\alpha, (\bar{x}_1 \vee x_3)_\beta, (\bar{x}_4 \vee \bar{x}_5)_\gamma\}$. *Before starting the execution, the propagation queue is empty* $Q = [\|]$. *We use the symbol* $\|$ *to separate propagated literals (on the left) from literals pending propagation (on the right). If literal* x_1 *is selected, it is added to* Q. *Before propagation the queue contains* $Q = [\|x_1^d]$. *UP will propagate* x_1 *and add two new consequences* x_2 *and* x_3. *The propagation queue is now* $Q = [x_1^d\|x_2(\alpha), x_3(\beta)]$ *and the current assignment is* $\{x_1, x_2, x_3\}$. *The propagation of* x_2 *and* x_3 *does not add new literals to Q, so it becomes* $Q = [x_1^d, x_2(\alpha), x_3(\beta)\|]$

If x_4 *is decided, UP will add a new consequence* $\bar{x}_5$. *After the propagation, we have* $Q = [x_1^d, x_2(\alpha), x_3(\beta), x_4^d, \bar{x}_5(\gamma)\|]$. *The current assignment is* $\{x_1, x_2, x_3, x_4, \bar{x}_5\}$. *Note that no more literals can be propagated and a complete assignment has been found. Note as well that Q contains two decision levels: the first one is formed by literals* x_1, x_2 *and* x_3 *while the second one is formed by literals* x_4 *and* $\bar{x}_5$.

3.1.1 LAZY DATA STRUCTURES.

As mentioned, the aim of UP is to detect all units and all conflicting clauses. Taking into account that this process typically takes up to 80% of the total runtime of a SAT solver, it is important to design efficient data structures.

The first attempt was the use of *adjacency lists*. For each literal, one keeps the list of all clauses in which the literal appears. Then, upon the addition of a literal l to the assignment, only clauses containing $\bar{l}$ have to be traversed. The main drawback of further refinements to detect efficiently when a clause has become unit, such as keeping *counters* indicating the number of unassigned literals of a clause, is that they involved a considerable amount of work upon backtracking.

The method used by MINISAT is the *two-watched literal scheme* introduced by Moskewicz et al. (2001). Its basic idea is that a clause cannot be unit or conflicting if (i) it has one satisfied literal or (ii) it has two unassigned literals.

The algorithm keeps two special literals for each clause, called the *watched literals*, initially two unassigned literals, and tries to maintain the invariant that always one satisfied literal or two unassigned literals are watched.

The invariant may be broken only if one of the two watched literals becomes falsified. In this case, the clause is traversed looking for another non-false literal to watch in order to restore the invariant. If one such literal cannot be found, the clause is declared to be true, unit or conflicting depending on the value of the other watched literal. Hence, when a literal l is added to the assignment, the clauses that may have become falsified or unit (line 14 in Algorithm 2) are only those clauses where $\bar{l}$ is watched.

The main advantage of such an approach is that no work on the clauses has to be done upon backtracking. However, the main drawback is that the only way to know how many literals are unassigned for a given clause is by traversing all its literals. Note that this information is used by other techniques such as the *Two-sided Jeroslow* branching heuristic (See Section 3.3).

3.1.2 RESOLUTION REFUTATION TREES.

If UP detects a conflict, an unsatisfiable subset of clauses $\mathcal{F}'$ can be determined using the information provided by Q. Since $\mathcal{F}'$ is unsatisfiable, the empty clause $\Box$ can be derived from $\mathcal{F}'$ via resolution. Such resolution process is called a *refutation*. A refutation for an unsatisfiable clause set $\mathcal{F}'$ is a *resolution refutation tree* (or simply a *refutation tree*) if every clause is used exactly once during the resolution process.

A refutation tree Υ can be built from the propagation queue Q as follows: let C_0 be the conflicting clause. Traverse Q in a *LIFO* (*Last In First Out*) fashion until a clashing clause D_0 is found. Then resolution is applied between C_0 and D_0, obtaining resolvent C_1. Next, the traversal of Q continues until a clause D_1 that clashes with C_1 is found, giving resolvent C_2 and we iterate the process until the resolvent we obtain is the empty clause $\Box$. The importance of refutation trees will become relevant in Section 6.

Example 2 *Consider* $\mathcal{F} = \{(\bar{x}_1)_\alpha, (x_1 \vee x_4)_\beta, (x_1 \vee x_2)_\gamma, (x_1 \vee x_3 \vee \bar{x}_4)_\delta, (x_1 \vee \bar{x}_2 \vee \bar{x}_3)_\varepsilon, (x_1 \vee \bar{x}_5)_\varphi\}$. *If we apply unit propagation the unit clause* α *is enqueued producing* $Q = [\|\bar{x}_1(\alpha)]$. *Then* $\bar{x}_1$ *is propagated and Q becomes* $[\bar{x}_1(\alpha)\|x_4(\beta), x_2(\gamma), \bar{x}_5(\varphi)]$. *After that, literal* x_4 *is propagated causing clause* δ *to become unit and Q becomes* $[\bar{x}_1(\alpha), x_4(\beta)\|x_2(\gamma), \bar{x}_5(\varphi), x_3(\delta)]$. *After that, literal* x_2 *is propagated and clause* ε *is found to be conflicting. Figure 1.a shows the state of Q after the propagation.*

Now we build the refutation tree. Starting from the tail of Q the first clause clashing with the conflicting clause ε *is* δ. *Resolution between* ε *and* δ *generates the resolvent* $x_1 \vee \bar{x}_2 \vee \bar{x}_4$. *The first clause clashing with* x_2 *is* γ*, producing resolvent* $x_1 \vee \bar{x}_4$. *The next clause clashing with* x_4 *is* β *and resolution generates* x_1. *Finally, we resolve with clause* α *and we obtain* $\Box$*.Figure 1.b shows the resulting refutation tree.*

3.2 Learning and Backjumping

Learning and backjumping are best illustrated with an example (see Silva & Sakallah, 1996; Zhang et al., 2001, for a precise description):

Example 3 *Consider the formula* $\{(\bar{x}_1 \vee x_2)_\alpha, (\bar{x}_3 \vee x_4)_\beta, (\bar{x}_5 \vee \bar{x}_6)_\gamma, (\bar{x}_2 \vee \bar{x}_5 \vee x_6)_\delta\}$ *and the partial assignment* $\{x_1, x_2, x_3, x_4, x_5, \bar{x}_6\}$ *that leads to a conflict over clause* δ. *Suppose that the current propagation queue is* $Q = [x_1^d, x_2(\alpha), x_3^d, x_4(\beta), x_5^d, \bar{x}_6(\gamma)\|]$.

In the example it is easy to see that decision x_1^d *is incompatible with decision* x_5^d. *Such incompatibility can be represented with clause* $(\bar{x}_1 \vee \bar{x}_5)$. *Similarly, consequence* x_2 *is incompatible with decision* x_5^d *and it can be represented with the clause* $(\bar{x}_2 \vee \bar{x}_5)$.

$$\mathcal{F} = \{(\bar{x}_1)_\alpha, (x_1 \vee x_4)_\beta, (x_1 \vee x_2)_\gamma, (x_1 \vee x_3 \vee \bar{x_4})_\delta, (x_1 \vee \bar{x}_2 \vee \bar{x}_3)_\varepsilon, (x_1 \vee \bar{x_5})_\varphi\}$$

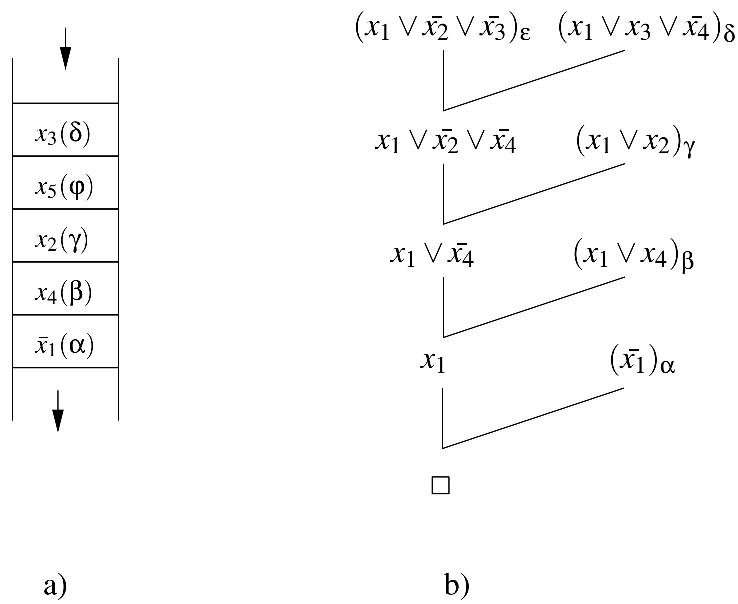


Figure 1: Graphical representation of the propagation queue Q and a refutation tree Υ of example 2. On the top, the original formula $\mathcal{F}$. On the left, the propagation Q after step 1. Arrows indicate the order in which resolving clauses are selected. On the right, the resolution tree computed in step 2.

Clause learning implements different techniques that are used to *discover* such implicit incompatibilities and adds them to the formula. Learned clauses can accelerate the subsequent search, since they can increase the potential of future UP executions. However, it has been observed that unrestricted clause learning can be impractical in some cases (recorded clauses consume memory and repeated recording may lead to its exhaustion). For this reason, current SAT solvers incorporate different clause deletion policies in order to remove some of the learned clauses.

Learned clauses can also be used to backjump if their presence would have allowed a unit propagation at an earlier decision level. In this case, we say that the clause is *asserting* and backjumping can proceed by going back to that level and adding the unit propagated literal. Among the several automated ways of generating asserting clauses, MINISAT uses the so-called *First Unique Implication Point* (*1UIP*) (Zhang et al., 2001).

3.3 Branching Heuristic

Branching occurs in the function `SelectLiteral` (Algorithm 1). When there are no more literals to propagate, this function chooses one variable from all the unassigned ones and assigns it a value.

The importance of the branching heuristic is well known, since different branching heuristic may produce different-sized search trees.

Early branching heuristics include the *Bohm's Heuristic* (Buro & Büning, 1993), the *Maximum Ocurrences on Minimum sized clauses* (MOM) (Freeman, 1995) and the *Two sided-Jeroslow Wang Heuristic* (Jeroslow & Wang, 1990). Those heuristics try to choose the literal such that its assignment will generate the largest number of implications or that satisfy most clauses. All these heuristics are *state dependent*, that is, they use information about the state of the clauses given the current assignment. In most of them, such information is the number of unassigned literals for each clause. Hence, they were implemented jointly with data structures based on adjacency lists since they keep such information. For instance, the Two sided-Jeroslow Wang Heuristic computes for each literal l of F the following function:

$$J(l) = \sum_{\substack{C \in F \\ s.t.\ l \in C}} 2^{-|C|}$$

and selects the literal l that maximizes function $J(l)$.

As solvers become more efficient, updating metrics of state-dependent heuristics dominates the execution time. Hence MINISAT uses a slight modification of a state-independent heuristic first proposed by Moskewicz et al. (2001). Such heuristic, called *Variable State Independent Decaying Sum* (*VSIDS*), selects the literal that appears more frequently over all clauses, but giving priority to recently learned clauses. The advantage of this heuristic is that metrics only have to be updated when clauses are learned. Since this only occurs occasionally, its computation has very low overhead. The VSIDS heuristic suits perfectly with lazy data structures such as the two-watched literal scheme.

4. (Weighted) Max-SAT

A *weighted* clause is a pair (C, w), where C is a clause and w is an integer representing the cost of its falsification, also called its *weight*. If a problem contains clauses that *must* be satisfied, we call such clauses *mandatory* or *hard* and associate with them a special weight $\top$. Non-mandatory clauses are also called *soft*. A *weighted formula* in *conjunctive normal form* (WCNF) is a set of weighted clauses. A *model* is a complete assignment that satisfies all mandatory clauses. The *cost of an assignment* is the sum of weights of the clauses that it falsifies. Given a WCNF formula $\mathcal{F}$, *Weighted* Max-SAT is the problem of finding a model of $\mathcal{F}$ of minimum cost. This cost will be called the *optimal cost of* $\mathcal{F}$. Note that if a formula contains only mandatory clauses, weighted Max-SAT is equivalent to classical SAT. If all the clauses have weight 1, we have the so-called (unweighted) Max-SAT problem. In the following, we will assume weighted Max-SAT.

We say that a weighted formula $\mathcal{F}'$ is a *relaxation* of a weighted formula $\mathcal{F}$ (noted $\mathcal{F}' \sqsubseteq \mathcal{F}$) if the optimal cost of $\mathcal{F}'$ is less than or equal to the optimal cost in $\mathcal{F}$ (non-models are considered to have cost infinity). We say that two weighted formulas $\mathcal{F}'$ and $\mathcal{F}$ are *equivalent* (noted $\mathcal{F}' \equiv \mathcal{F}$) if $\mathcal{F}' \sqsubseteq \mathcal{F}$ and $\mathcal{F} \sqsubseteq \mathcal{F}'$.

Max-SAT simplification rules transforms a formula $\mathcal{F}$ into an equivalent, but presumably simpler formula $\mathcal{F}'$. All SAT simplification rules (e.g. unit propagation, tautology removal,...) can be directly applied to Max-SAT if restricted to mandatory clauses. However, several specific Max-SAT simplification rules exist (Larrosa et al., 2007). For instance, if a formula contains clauses (C, u) and (C, v), they can be replaced by $(C, u+v)$. If it contains a clause $(C, 0)$, it may be removed. If it contains a unit $(l, \top)$, it can be simplified by removing all (including soft) clauses containing l and removing $\bar{l}$ from all the clauses (including soft clauses) where it appears. The application of this rule until quiescence is the natural extension of *unit propagation* to Max-SAT.

The empty clause may appear in a weighted formula. If its weight is $\top$, it is clear that the formula does not have any model. If its weight is $w < \top$, the cost of any assignment will include that weight, so w is an obvious lower bound of the formula optimal cost. Weighted empty clauses and their interpretation in terms of lower bounds will become relevant in Section 6.

As shown by Larrosa et al. (2007), the notion of resolution can be extended to weighted formulas as follows [1] ,

$$\{(x \vee A, u), (\bar{x} \vee B, w)\} \equiv \left\{ \begin{array}{l} (A \vee B, m), \\ (x \vee A, u - m), \\ (\bar{x} \vee B, w - m), \\ (x \vee A \vee \bar{B}, m), \\ (\bar{x} \vee \bar{A} \vee B, m) \end{array} \right\}$$

where A and B are arbitrary disjunctions of literals and $m = \min\{u, w\}$.

$(x \vee A, u)$ and $(\bar{x} \vee B, w)$ are called the *prior clashing clauses*, $(A \vee B, m)$ is called the *resolvent*, $(x \vee A, u - m)$ and $(\bar{x} \vee B, w - m)$ are called the *posterior clashing clauses*, and $(x \vee A \vee \bar{B}, m)$ and $(\bar{x} \vee \bar{A} \vee B, m)$ are called the *compensation clauses*. The effect of Max-SAT resolution, as in classical resolution, is to infer (namely, make explicit) a connection between A and B. However, there is an important difference between classical resolution and Max-SAT resolution. While the former yields the *addition* of a new clause, Max-RES is a transformation rule. Namely, it requires the *replacement* of the left-hand clauses by the right-hand clauses. The reason is that some cost of the prior clashing clauses must be substracted in order to *compensate* the new inferred information. Consequently, Max-RES is better understood as a *movement* of knowledge in the formula.

The resolution rule for Max-SAT preserves equivalence ($\equiv$). The last two compensation clauses may lose the clausal form, so the following rule (Larrosa et al., 2007) may be needed to recover it:

$$CNF(A \vee \overline{l \vee B}, u) = \left\{ \begin{array}{rcl} A \vee \bar{l} & : & |B| = 0 \\ \{(A \vee \bar{l} \vee B, u)\} \cup CNF(A \vee \bar{B}, u) & : & |B| > 0 \end{array} \right.$$

Example 4 If we apply weighted resolution to the following clauses $\{(x_1 \vee x_2, 3), (\bar{x}_1 \vee x_2 \vee x_3, 4)\}$ we obtain $\{(x_2 \vee x_2 \vee x_3, 3), (x_1 \vee x_2, 3-3), (\bar{x}_1 \vee x_2 \vee x_3, 4-3), (x_1 \vee x_2 \vee \overline{(x_2 \vee x_3)}, 3), (\bar{x}_1 \vee \bar{x}_2 \vee x_2 \vee x_3, 3)\}$. The first clause can be simplified. The second clause can be omitted because it weight is zero. The fifth clause can be omitted because it is a tautology. The fourth element is not a clause because it is not a simple disjunction. Hence, we apply CNF rule to it and we obtain two new clauses $CNF(x_1 \vee x_2 \vee \overline{(x_2 \vee x_3)}, 3) = \{(x_1 \vee x_2 \vee \bar{x}_2 \vee x_3, 3), (x_1 \vee x_2 \vee \bar{x}_3, 3)\}$. Note that the first new clause is a tautology. Therefore, we obtain the equivalent formula $\{(x_2 \vee x_3, 3), (\bar{x}_1 \vee x_2 \vee x_3, 1), (x_1 \vee x_2 \vee \bar{x}_3, 3)\}$.

5. Overview of MINIMAXSAT

MINIMAXSAT is a weighted Max-SAT solver built on top of MINISAT+ (Eén & Sörensson, 2006). Any other DPLL-based SAT solver could have been used, but MINISAT+ was particularly well-suited because of its short and open-source code. Besides, it can deal with pseudo-boolean constraints.

1. If A is the empty clause then $\bar{A}$ represents a tautology. For the special weight $\top$, we have the relations $\top - m = \top$ and $\top - \top = \top$ (Larrosa et al., 2007)

Algorithm 3: MINIMAXSAT basic structure.

```
Function Search() : integer
    ub := LocalSearch(); lb := 0 ;
    InitQueue(Q) ;
    Loop
        Propagate() ;
        if Hard Conflict then
            AnalyzeConflict() ;
            if Top Level Hard Conflict then return ub ;
            else
                LearnClause() ;
                Backjump() ;
        else if Soft Conflict then
            ChronologicalBactrack() ;
            if End of Search then return ub ;
        else if all variables assigned then
            ub := lb ;
            if ub = 0 then return ub ;
            ChronologicalBactrack() ;
            if End of Search then return ub ;
        else
            l := SelectLiteral() ;
            Enqueue(Q, l) ;
```

Given a WCNF formula (possibly containing hard and soft clauses), MINIMAXSAT returns the cost of the optimal model (or $\top$ if there is no model). This is achieved by means of a branch-and-bound search, as it is usually done to solve optimization problems.

Like MINISAT, the tree of assignments is traversed in a depth-first manner. At each search point, the algorithm tries to simplify the current formula and, ideally, detect a conflict, which would mean that the current partial assignment cannot be successfully extended. MINIMAXSAT distinguishes two types of conflicts: hard and soft. *Hard* conflicts indicate that there is no model extending the current partial assignment (namely, all the mandatory clauses cannot be simultaneously satisfied). Hard conflicts are detected taking only into account hard clauses and using the methods of MINISAT. When a hard conflict occurs, MINIMAXSAT learns a hard clause and backjumps as MINISAT would do. *Soft* conflicts indicate that the current partial assignment cannot be extended to an optimal assignment. In order to identify soft conflicts, the algorithm maintains two values during the search:

- The cost of the best model found so far, which is an upper bound ub of the optimal solution.
- An underestimation of the best cost that can be achieved extending the current partial assignment into a model, which is a lower bound lb of the current subproblem.

A soft conflict is detected when $lb \geq ub$, because it means that the current assignment cannot lead to an optimal model. When a soft conflict is detected, the algorithm backtracks chronologically. Note

Algorithm 4: MiniMaxSat propagation.

Function *MS-UP() : conflict*
- **while** (Q *contains non-propagated literals*) **do**
  - $l :=$ GetFirstNonPropagatedLit(Q); MarkAsPropagated(l) ;
  - $lb := lb + V(\bar{l})$) ;
  - **if** $lb \geq ub$ **then return** *Soft Conflict* ;
  - **foreach** *Hard clause* $(C \vee \bar{l}, \top)$ *that becomes unit or falsified* **do**
    - **if** $(C \vee \bar{l}, \top)$ *becomes unit* $(q, \top)$ **then** Enqueue(Q, q) ;
    - **else if** $(C \vee \bar{l}, \top)$ *becomes falsified* **then return** *Hard Conflict* ;
  - **foreach** *Soft clause* $(C \vee \bar{l}, u)$ *that becomes unit* **do**
    - **if** $(C \vee \bar{l}, u)$ *becomes a unit* (q, u) **then** $V(q) := V(q) + u$;
- **return** *None* ;

Function *Propagate() : conflict*
- $c :=$ MS-UP( ) ;
- **if** $c =$ *Hard or Soft Conflict* **then return** c ;
- improveLB( ) ;
- **if** $lb \geq ub$ **then return** *Soft Conflict* ;
- **return** *None* ;

that one could also backjump by computing a clause expressing the reasons that led to $lb \geq ub$. However, in the presence of lots of soft clauses, this approach ends up creating too many long clauses which affect negatively to the efficience of the solver and hence we decided to perform simple chronological backtracking.

We also want to remark that any soft clause (C, w) with $w \geq ub$ must be satisfied in an optimal assignment. Therefore, in the following we assume that such soft clauses are automatically transformed into hard clauses previous to search. Other than those ones, no other soft clause is promoted into a hard one during the search.

An algorithmic description of MINIMAXSAT is presented in *Algorithm 3*. Before starting the search, a good initial upper bound is obtained with a local search method (line 17) which may yield the identification of some new hard clauses. In our current implementation we use UBCSAT (Tompkins & Hoos, 2004) with default parameters. The selected local search algorithm is *IROTS* (*Iterated Robust Tabu Search*) (Smyth, Hoos, & Stützle, 2003). Besides, the lower bound is initialized to zero. Next, the queue Q is initialized with all unit hard clauses in the resulting formula (line 18). The main loop starts in line 19 and each iteration is in charge of propagating all pending literals (line 20) and, if no conflict is detected, attempting the extension of the current partial assignment (line 26). Pending literals in Q are propagated in function `Propagate` (line 20), which may return a hard or soft conflict. If a hard conflict is encountered (line 21) the conflict is analyzed, a new hard clause is learned and backjumping is performed. This is done as introduced in Section 3. If a soft conflict is encountered (line 22) chronological backtracking is performed. If no conflict is found (line 26), a literal is heuristically selected and added to Q for propagation in the next iteration. However, if the current assignment is complete (line 23), the upper bound is updated. Search stops if a zero-cost solution is found, since it cannot be further improved (line 24). Else, chronological backtracking is performed (line 25). Note that backjumping leads to termination if a top level hard conflict is found, while chronological backtracking leads to termination if the two values for the first assigned variable have been tried.

Algorithm 4 describes the propagation process (function `Propagate`). It uses an array $V(l)$ which accumulates the weight of all soft clauses that have become unit over l; namely, original clauses $(A \vee l, w)$ such that the current assignment falsifies A. If no such clauses exists, we assume $V(l) = 0$. First of all, it performs a Max-SAT-adapted form of unit propagation (`MS-UP`, line 35). `MS-UP` iterates over the non-propagated literals l in Q (line 27). Firstly, adding l to the assignment may make a set of soft clauses falsified. Since the cost of all such clauses is kept in $V(\bar{l})$, we add it to the lower bound (line 28). If the lower bound increment identifies a soft conflict, it is returned (line 29). Then, if a hard clause becomes unit, the corresponding literal is added to Q for future propagation (line 31). Finally, if a soft clause becomes a unit clause (q, u) (line 33), its weight u is added to $V(q)$ (line 34). If during this process a hard conflict is detected, the function returns it (lines 32,36). Else, the algorithm attempts to detect a soft conflict with a call to procedure `improveLB` (line 37), and it returns the soft conflict if it is found (line 38). In the next section a detailed description of `improveLB` can be found. Finally, if no conflict is detected, the function returns *None* (line 39).

6. Lower Bounding in MINIMAXSAT

In the following, we consider an arbitrary search state of MINIMAXSAT before the call to the procedure `improveLB`. For the purpose of this section, such a search state can be characterized by the current assignment. The current assignment determines the *current subformula* which is the original formula *conditioned* by the current assignment: If a clause contains a literal that is part of the current assignment, it is removed. Besides, all the literals whose negation appear in the current assignment are removed from the clauses where they appear.

The value of *lb* maintained by MINIMAXSAT is precisely the aggregation of costs of all the clauses that have become empty due to the current assignment. Similarly, we recall that the value $V(l)$ is the aggregation of costs of all the clauses that have become unit over l due to the current assignment. Thus, the current subformula contains $(\square, lb)$ and $(l, V(l))$ for every l.

MINIMAXSAT computes its lower bound by deriving new soft empty clauses $(\square, w)$ through a resolution process. Such clauses are added to the already existing clause $(\square, lb)$ producing an increment of the lower bound.

As a first step, `improveLB` replaces each occurrence of (l, u) and $(\bar{l}, w)$ by $(l, u - m), (\bar{l}, w - m), (\square, m)$ (with $m = \min\{u, w\}$), which amounts to applying a restricted version of Max-SAT resolution known as Unit Neighborhood Resolution (UNR) (Larrosa et al., 2007).

It produces an immediate increment of the lower bound (*i.e.*, the weight of the empty clause at line 43) as it is illustrated in the following example,

Example 5 Consider the current state is $\{(\square, 3), (x_1, 1), (x_2, 1), (\bar{x}_1, 2), (\bar{x}_2, 2), (x_1 \vee x_2, 3)\}$. UNR would resolve on clauses $(x_1, 1)$ and $(\bar{x}_1, 2)$ replacing them by $(\bar{x}_1, 1)$ and $(\square, 1)$ (all other compensation clauses are removed because their weight is zero or they are tautologies). The two empty clauses can be grouped into $(\square, 3 + 1 = 4)$. UNR would also resolve on clauses $(x_2, 1)$ and $(\bar{x}_2, 2)$ replacing them by $(\bar{x}_2, 1)$ and $(\square, 1)$. The two empty clauses can be grouped into $(\square, 4 + 1 = 5)$. So, the new equivalent formula is $\{(\square, 5), (\bar{x}_1, 1), (\bar{x}_2, 1), (x_1 \vee x_2, 3)\}$ with a higher lower bound of 5.

Algorithm 5: Lower Bounding in MINIMAXSAT

Function *SUP() : conflict*
40 InitQueue(Q) ;
while (Q *contains non-propagated literals*) **do**
$l :=$ GetFirstNonPropagatedLit(Q); MarkAsPropagated(l) ;
41 **foreach** *(Hard or Soft) Clause* $C \vee \bar{l}$ *that becomes unit or falsified* **do**
if $C \vee \bar{l}$ *becomes a unit* q **then** Enqueue(Q, q) ;
else if $C \vee \bar{l}$ *becomes falsified* **then return** *conflict* ;
return *None* ;

Procedure *improveLB() : lb*
42 **foreach** $(l, v), (\bar{l}, w) \in F$ **do**
43 replace them by $(l, v-m), (\bar{l}, w-m), (\Box, m)$ with $m := \min(v, w)$;
44 **while** $SUP() = conflict$ **do**
45 $\Upsilon :=$ BuildTree() ;
46 $m :=$ minimum weight among clauses in Υ;
47 **if** *Condition* **then** ApplyResolution(Υ, m) ;
48 **else** $lb := lb + m$; remove weight m from clauses in Υ;

As a second step `improveLB` executes a *simulation of unit propagation* (`SUP`, line 44) in which soft clauses are treated as if they were hard. First, `SUP` adds to Q all unit soft clauses (line 40). Then, the new literals in Q are propagated. When new (hard or soft) clauses become unit, they are inserted in Q (line 41). If `SUP` yields a conflict, it means that there is a subset of (soft or hard) clauses that cannot be simultaneously satisfied. We showed in Section 3 that Q can be used to identify such subset and build a refutation tree Υ. `ImproveLB` computes such a tree (line 45). If we take into account again the weights of the clauses and apply Max-SAT resolution (Section 4) as dictated by Υ, one can see that it will produce a new clause $(\Box, m)$, where m is the minimum weight among all the clauses in the tree (line 46). It means that the extension of the current partial assignment to the unassigned variables will have a cost of at least m.

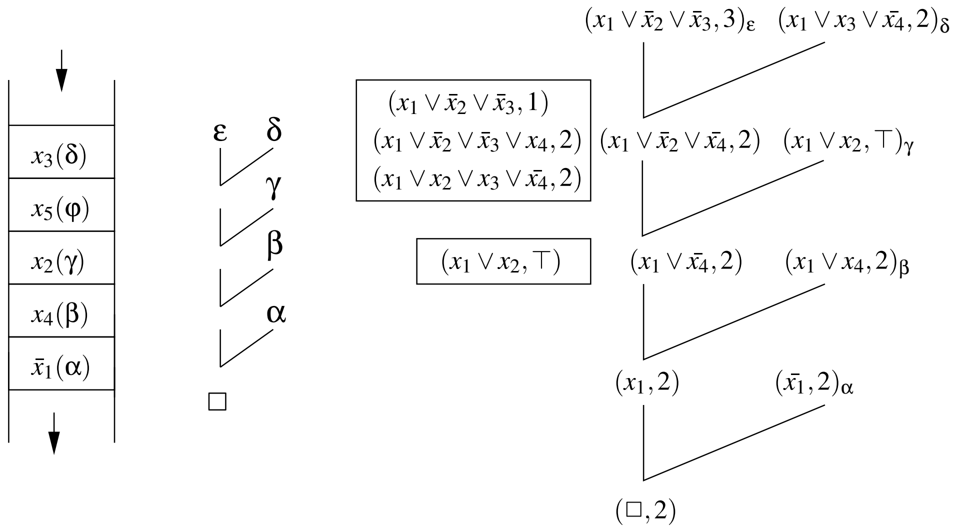
It is important to remark that at each step in the Max-SAT resolution process we do not consider the minimum of the weight of the two clauses, but rather the minimum of all the clauses in the resolution tree. This is why m is passed as a parameter in line 47.

The result of the resolution process is the replacement of all the clauses in the leaves of Υ by $(\Box, m)$ and the corresponding compensation clauses (function `ApplyResolution` in line 47), thus obtaining an equivalent formula with a lower bound increment of m. We call this procedure *resolution-based* lower bounding.

Example 6 Consider the formula $\mathcal{F} = \{(\bar{x}_1, 2)_\alpha, (x_1 \vee x_4, 1)_\beta, (x_1 \vee x_2, \top)_\gamma, (x_1 \vee x_3 \vee \bar{x}_4, 2)_\delta, (x_1 \vee \bar{x}_2 \vee \bar{x}_3, 3)_\varepsilon, (x_1 \vee \bar{x}_5, 1)_\varphi\}$

Step 1. *Apply `SUP`.* Initially, the unit clause α is enqueued producing $Q = [\|\bar{x}_1(\alpha)]$. Then $\bar{x}_1$ is propagated and Q becomes $[\bar{x}_1(\alpha)\|x_4(\beta), x_2(\gamma), \bar{x}_5(\varphi)]$. Literal x_4 is propagated and clause δ becomes unit, producing $Q = [\bar{x}_1(\alpha), x_4(\beta)\|x_2(\gamma), \bar{x}_5(\varphi), x_3(\delta)]$. After that, literal x_2 is propagated and clause ε is found to be conflicting. Figure 2.a shows the state of Q after the propagation.

$$\mathcal{F} = \{(\bar{x}_1, 2)_\alpha, (x_1 \vee x_4, 2)_\beta, (x_1 \vee x_2, \top)_\gamma, (x_1 \vee x_3 \vee \bar{x_4}, 2)_\delta, (x_1 \vee \bar{x}_2 \vee \bar{x}_3, 3)_\varepsilon, (x_1 \vee \bar{x_5}, 1)_\varphi\}$$



a) b) c)

$$\mathcal{F}' = \{(x_1 \vee x_2, \top), (x_1 \vee \bar{x_5}, 1), (\square, 2), (x_1 \vee \bar{x_2} \vee \bar{x_3}, 1), (x_1 \vee \bar{x_2} \vee \bar{x_3} \vee x_4, 2), (x_1 \vee x_2 \vee x_3 \vee \bar{x_4}, 2)\}$$
$$\mathcal{F}'' = \{(x_1 \vee x_2, \top), (x_1 \vee \bar{x_2} \vee \bar{x_3}, 1), (x_1 \vee \bar{x_5}, 1), (\square, 2)\}$$

Figure 2: Graphical representation of MINIMAXSAT lower bounding. On the top, the original current formula $\mathcal{F}$. On the left, the propagation Q after step 1. In the middle, the structure of the refutation tree computed by the simulation of UP in step 2. On the right, the effect of actually executing the Max-SAT resolution (step 3). The resulting formula $\mathcal{F}'$ appears bellow. If substraction-based lower bounding is performed, step 3 is replaced by a substraction of weights, producing formula $\mathcal{F}''$.

Step 2. *Build the simulated refutation tree.* Starting from the tail of Q the first clause clashing with the conflicting clause ε is δ. Resolution between ε and δ generates the resolvent $x_1 \vee \bar{x}_2 \vee \bar{x_4}$. The first clause clashing with x_2 is γ, producing resolvent $x_1 \vee \bar{x_4}$. The next clause clashing with x_4 is β and resolution generates x_1. Finally, we resolve with clause α and we obtain □.Figure 2.b shows the resulting resolution tree.

Step 3. *Apply Max-SAT resolution.* We apply Max-SAT resolution as indicated by the refutation tree computed in Step 2. Figure 2.c graphically shows the result of the process. Leaf clauses are the original (weighted) clauses involved in the resolution. Each internal node indicates a resolution step. The resolvents appear in the junction of the edges. Beside each resolvent, inside a box, there are the compensation clauses that must be added to the formula to preserve equivalence. Since clauses that are used in resolution must be removed, the resulting formula $\mathcal{F}'$ consists of the root of the tree $((\square,2))$,all compensation clauses and all clauses not used in the refutation tree. That is, the resulting formula is $\mathcal{F}' = \{(x_1 \vee x_2, \top), (x_1 \vee \bar{x_5}, 1), (\square, 2), (x_1 \vee \bar{x_2} \vee \bar{x_3}, 1), (x_1 \vee \bar{x_2} \vee \bar{x_3} \vee x_4, 2), (x_1 \vee x_2 \vee x_3 \vee \bar{x_4}, 2)\}$. The soundness of Max-SAT resolution guarantees that $\mathcal{F} \equiv \mathcal{F}'$.

Remark 1 *All the transformations applied by the* resolution-based lower bounding *can be passed on to descendent nodes because the changes preserve equivalence. Nevertheless, transformations have to be restored when backtracking takes place.*

An alternative to problem transformation through resolution is to identify the lower bound increment m and then substract it from all the clauses that would have participated in the resolution tree. This procedure is similar to the lower bound computed by Li et al. (2005) and we call it *substraction-based* (line 48) lower bounding.

Example 7 Consider formula $\mathcal{F}$ from the previous example. Steps 1 and 2 are identical. However, substraction-based lower bounding would replace Step 3 by Step 3' that substracts weight 2 from the clauses that appear in the refutation tree and then adds $(\square, 2)$ to the formula. The result is $\mathcal{F}'' = \{(x_1 \vee x_2, \top), (x_1 \vee \bar{x_2} \vee \bar{x_3}, 1), (x_1 \vee \bar{x_5}, 1), (\square, 2)\}$. Note that $\mathcal{F}'' \sqsubseteq \mathcal{F}$.

Remark 2 *All the substractions applied by the* substraction-based lower bounding *have to be restored before moving to a descendent node because they do not preserve equivalence.*

After the increment of the lower bound with either technique, procedure `SUP` can be executed again, which may yield new lower bound increments. The process is repeated until `SUP` does not detect any conflict.

When comparing the two previous approaches, we observe that resolution-based lower bounding has a larger overhead, because resolution steps need to be actually computed and their consequences must be added to the current formula and removed upon backtracking. However, the effort invested in the transformation may be well amortized because the increment obtained in the lower bound *becomes part of the current formula*, so it does not have to be *discovered* again and again by all the descendent nodes of the search. On the other hand, substraction-based lower bounding has a smaller overhead because resolution needs not to be actually computed. This also facilitates the context restoration upon backtracking.

MINIMAXSAT incorporates the two alternatives and chooses to apply one or the other heuristically (lines 47,48) depending on a specific *condition* (line 47). We observed that resolution-based lower bounding seems to be more effective if resolution is only applied to low arity clauses. As a consequence, after the identification of the resolution tree, MINIMAXSAT applies resolution-based lower bounding only if the largest resolvent in the resolution tree has arity strictly less than 4. Otherwise, it applies substraction-based lower bounding. See Section 8 for more details.

7. Additional Features of MINIMAXSAT

In this section we overview other important features of MINIMAXSAT, namely the use of the two-watched literal scheme, its branching heuristic, the use of soft probing and how MINIMAXSAT deals with pseudo-boolean functions.

7.1 Two-Watched Literals

MINIMAXSAT uses the *two-watched literal scheme* also on soft clauses. Recall that one of the main advantages of this technique, when applied to pure SAT problems, is that when backtracking takes place, no work has to be done on the clauses. Unfortunately, in the case of soft clauses some restoration needs to be done. When a soft clause becomes unit over literal l in function *MS-UP*, its weight is added to $V(l)$ and the clause is eliminated (or marked as eliminated) to avoid reusing it in the lower bounding procedure. These changes, as well as any addition to lb, have to be restored when backtracking is performed. However, note that during the executions of *SUP* (simulation of unit propagation) all clauses are considered as hard. In this case the two-watched literal scheme works exactly as in a SAT solver with both hard and soft clauses. When an inconsistency is detected by *SUP* or it stops because there are no more literals to propagate, the initial state has to be recovered. In that situation restoring the initial state is completely overhead free.

7.2 Branching Heuristic

MINIMAXSAT incorporates two alternative branching heuristics. The first one is the VSIDS heuristic (Moskewicz et al., 2001) disregarding soft clauses (that is, MINISAT'S default). This heuristic is likely to be good in structured problems in which learning and backjumping play a significant role, as well as in problems in which it is difficult to find models (namely, the satisfaction component of the problem is more difficult than the optimization component). Since this heuristic disregards soft clauses, it is likely to be ineffective in problems where it is easy to find models and the difficulty is to find the optimal one and prove its optimality. In the extreme case, where problems only contain soft clauses (every complete assignment is a model) the VSIDS heuristic is blind and therefore completely useless.

To overcome this limitation of VSIDS, MINIMAXSAT also incorporates the *Weighted Jeroslow* heuristic (Heras & Larrosa, 2006). It is the extension of the SAT Jeroslow heuristic described in Section 3. Given a weighted formula F, for each literal l of F the following function is defined:

$$J(l) = \sum_{\substack{(C,w)\in F \\ s.t.\ l\in C}} 2^{-|C|} \cdot w$$

where mandatory clauses are assumed to have a weight equal to the upper bound ub. The heuristic selects the literal with the highest value of $J(l)$. Its main disadvantage is that metrics need to be updated at each visited node. In combination with the two-watched literal this updating becomes expensive and does not seem to pay off in general. Thus, in our current implementation of the heuristic, the $J(l)$ values are computed only at the root node and used throughout all the solving process. We found in our experiments that this heuristic is a good alternative in problems where the difficulty lies on the optimization part (e.g. problems with many models). MINIMAXSAT automatically changes from VSIDS to weighted Jeroslow if the problem does not contain any literal l such that there are some hard clauses with l and some other hard clauses with $\bar{l}$.

In both heuristics, if there is some literal l such that $V(l) + lb \geq ub$ at some node of the search tree, then $\bar{l}$ is the selected literal and l is never assigned.

7.3 Soft Probing

Probing is a well-known SAT technique that allows the formulation of hypothetical scenarios (Lynce & Silva, 2003). The idea is to temporarily assume that l is a hard unit clause and then execute unit propagation. If UP yields a conflict, we know that any model extending the current assignment must contain $\bar{l}$. The process is iterated over all the literals until quiescence. Exhaustive experiments in the SAT context indicate that it is too expensive to probe during the search (Le Berre, 2001; Lynce & Silva, 2003), so it is normally done as a pre-process in order to reduce the initial number of branching points.

We can easily extend this idea to Max-SAT. In that context, besides the *discovery* of unit hard clauses, it may be used to make explicit weighted unit clauses. We call it *soft probing*. As in SAT, the idea is to temporarily assume that l is a unit clause and then *simulate* unit propagation (*i.e.*, execute `SUP`()). Then, we build the resolution tree Υ from the propagation queue Q. If all the clauses in Υ are hard, we know that $\bar{l}$ must be added to the assignment. Else, we can reproduce Υ applying Max-SAT resolution with the weighted clauses and derive a unit clause $(\bar{l}, m)$ where m is the minimum weight among the clauses in Υ. Having unit soft clauses upfront makes the future executions of `improveLB` much more effective in the subsequent search. Besides, if we derive both (l,u) and $(\bar{l},w)$, we can generate via unit neighborhood resolution (see Example 5) an initial non-trivial lower bound of $min\{u,w\}$. We tested soft probing during the search and as a preprocessing in several benchmarks. We observed empirically that soft probing as a preprocessing was the best option as it is in SAT.

Example 8 Consider formula $\mathcal{F} = \{(x_1 \vee x_2, 1)_\alpha, (x_1 \vee x_3, 1)_\beta, (\bar{x}_2 \vee \bar{x}_3, 1)_\gamma\}$. If we assume $\bar{x}_1$ by adding it to Q and then execute `SUP` a conflict is reached. We obtain $Q = [\bar{x}_1^d, x_2(\alpha), x_3(\beta)]$ and we detect that γ is a conflicting clause. The clauses involved in the refutation tree are γ, β, and α. Resolving clauses γ and β results in $\{(x_1 \vee x_2, 1)_\alpha, (x_1 \vee \bar{x}_2, 1), (x_1 \vee x_2 \vee x_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, 1)\}$. The resolution of the previous resolvent and α produces the (equivalent) formula $\mathcal{F}' = \{(x_1, 1), (x_1 \vee x_2 \vee x_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, 1)\}$.

7.4 Pseudo-boolean Functions

A *pseudo-boolean optimization problem* (*PBO*) (Barth, 1995; Sheini & Sakallah, 2006; Eén & Sörensson, 2006) has the form:

minimize $\sum_{j=1}^{n} c_j \cdot x_j$

subject to $\sum_{j=1}^{n} a_{ij} l_j \geq b_i, \quad i = 1 \ldots m$

where $x_j \in \{0,1\}$, l_j is either x_j or $1 - x_j$, and c_j, a_{ij} and b_i are non-negative integers.

If MINIMAXSAT is provided with a PBO instance, it translates it into a Max-SAT formula as follows: each pseudo boolean constraint is translated into a set of *hard clauses* using MINISAT+ (Eén & Sörensson, 2006) (the algorithm heuristically decides the most appropriate translation choosing among *adders*, *sorters* or *BDDs*). The objective function is translated into a set of *soft unit clauses*. Each summand $c_j \cdot x_j$ becomes a new soft unit clause $(\bar{x}_j, c_j)$. After the translation MINIMAXSAT is executed as usual.

8. Empirical Results

In this section we present the benchmarks and the solvers used in our empirical evaluation. Then, we report the experiments performed in order to adjust the parameters of MINIMAXSAT. Finally, a comparison with other solvers is presented.

8.1 Benchmarks and Encodings

Having a good set of problems is fundamental to show the effectiveness of new solvers. In the following, we present several problems and we explain how to encode them as Weighted Max-SAT.

8.1.1 MAX-K-SAT

A k-SAT CNF formula is a CNF formula in which all clauses have size k. We generated random unsatisfiable 2-SAT and 3-SAT formulas with the *Cnfgen* generator[2] and solved the corresponding MAX-SAT problem. In the benchmarks, we fixed the number of variables and varied the number of clauses, which can be repeated.

8.1.2 MAX-CUT

Given a graph $G = (V,E)$, a *cut* is defined by a subset of vertices $U \subseteq V$. The size of a cut is the number of edges (v_i, v_j) such that $v_i \in U$ and $v_j \in V - U$. The *Max-cut* problem consists on finding a cut of maximum size. It can be encoded as Max-SAT associating one variable x_i to each graph vertex. Value *true* (respectively, *false*) indicates that vertex v_i belongs to U (respectively, to $V - U$). For each edge (v_i, v_j), there are two soft clauses $(x_i \vee x_j, 1), (\bar{x}_i \vee \bar{x}_j, 1)$. Given a complete assignment, the number of violated clauses is $|E| - S$ where S is the size of the cut associated to the assignment. In our experiments we considered Max-Cut instances extracted from random graphs of 60 nodes with varying number of edges.

8.1.3 MAX-ONE

Given a satisfiable CNF formula, *max-one* is the problem of finding a model with a maximum number of variables set to true. This problem can be encoded as Max-SAT by considering the clauses in the original formula as mandatory and adding a weighted unary clause $(x_i, 1)$ for each variable in the formula. Note that solving this problem is much harder than solving the usual SAT problem, because the search cannot stop as soon as a model is found. The optimal model must be found and its optimality must be proved. We considered the max-one problem over two types of CNF formula: random 3-SAT instances of 120 variables (generated with *Cnfgen*), and structured satisfiable instances coming from the 2002 SAT Competition[3].

8.1.4 MINIMUM VERTEX COVERING AND MAX-CLIQUE

Given a graph $G = (V,E)$, a *vertex covering* is a set $U \subseteq V$ such that for every edge (v_i, v_j) either $v_i \in U$ or $v_j \in U$. The size of a vertex covering is $|U|$. The *minimum vertex covering* problem consists in finding a covering of minimal size. It can be naturally formulated as (weighted) Max-SAT. We associate one variable x_i to each graph vertex v_i. Value *true* (respectively, *false*) indicates

2. A. van Gelder ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances
3. http://www.satcompetition.org/2002/

that vertex v_i belongs to U (respectively, to $V-U$). There is a binary hard $(x_i \vee x_j, \top)$ for each edge (v_i, v_j). It specifies that one or both of these two vertices have to be in the covering because there is an edge connecting them. There is a unary clause $(\bar{x}_i, 1)$ for each variable x_i, in order to specify that it is preferred not to add vertices to U. There is a simple way to transform minimum vertex coverings into max-cliques and vice-versa (Fahle, 2002).

In our experiments, we considered maximum clique instances extracted from random graphs with 150 nodes and varying number of edges. We also considered the 66 Max-Clique instances from the DIMACS challenge[4].

8.1.5 COMBINATORIAL AUCTIONS

A *combinatorial auction* is defined by a set of goods G and a set of bidders that bid for indivisible subsets of goods. Each bid i is defined by the subset of requested goods $G_i \subseteq G$ and the amount of money offered. The bid-taker, who wants to maximize its revenue, must decide which bids are to be accepted. Note that if two bids request the same good, they cannot be jointly accepted (Sandholm, 1999). In its Max-SAT encoding, there is one variable x_i associated to each bid. There are unit clauses (x_i, u_i) indicating that if bid i is not accepted there is a loss of profit u_i. Besides, for each pair i, j of conflicting bids, there is a mandatory clause $(\bar{x}_i \vee \bar{x}_j, \top)$.

In our experiments, we used the CATS generator (K. Leyton-Brown & Shoham, 2000) that allows to generate random instances inspired from real-world scenarios. In particular, we generated instances from the *Regions*, *Paths* and *Scheduling* distributions. The number of goods was fixed to 60 and we increased the number of bids. By increasing the number of bids, instances become more constrained (namely, there are more conflicting pairs of bids) and harder to solve.

8.1.6 MISCELLANEOUS

We also considered the following sets of instances widely used in the literature:

- The unsatisfiable instances of the 2nd DIMACS Implementation Challenge [5] considered by de Givry, Larrosa, Meseguer, and Schiex (2003) and Li et al. (2005): random 3-SAT instances (aim and dubois), pigeon hole problem (hole) and coloring problems (pret). Observe that all these instances are modelled as unweighted Max-SAT (i.e. all clauses have weight 1).

- *Max-CSP* random instances generated using the protocol specified by Larrosa and Schiex (2003) and de Givry, Heras, Larrosa, and Zytnicki (2005). We distinguish 4 different sets of problems: *Dense Loose* (*DL*), *Dense Tight* (*DT*), *Sparse Loose* (*SL*) and *Sparse Tight* (*ST*). Tight instances have about 20 variables while loose instances have about 40 variables. Each set contains 10 instances with 3 values and 10 instances with 4 values per variable.

- *Planning* (Cooper, Cussat-Blanc, de Roquemaurel, & Régnier, 2006) and *graph coloring* [6] structured instances taken from a *Weighted Constraint Satisfaction Problem* (*WCSP*) repository [7].

4. ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique
5. http://mat.gsia.cmu.edu/challenge.html
6. http://mat.gsia.cmu.edu/COLORING02/benchmarks
7. http://mulcyber.toulouse.inra.fr/plugins/scmcvs/cvsweb.php/benchs/?cvsroot=toolbar

- Problems taken from the 2006 pseudo-boolean evaluation [8]: *logic synthesis*, *misc* (*garden*), *routing*, *MPI* (*Minimum Prime Implicant)*, *MPS* (*miplib*). These instances are encoded to Max-SAT as specified in the previous section.

Note that Max-CSP, Planning and graph coloring instances are encoded into Max-SAT using the *direct encoding* (Walsh, 2000).

8.2 Alternative Solvers

We compare MINIMAXSAT with several optimizers from different communities. We restricted our comparison to freely available solvers. We considered the following ones:

- MAXSATZ (Li et al., 2006; Li, Manyà, & Planes, 2007). Unweighted Max-SAT solver. It was the best unweighted Max-SAT solver in the 2006 Max-SAT Evaluation.

- MAX-DPLL (Heras & Larrosa, 2006; Larrosa et al., 2007). Weighted Max-SAT solver. It is part of the TOOLBAR package. It was the best solver for weighted Max-SAT and the second best solver for unweighted Max-SAT in the 2006 Max-SAT Evaluation.

- TOOLBAR (Larrosa, 2002; Larrosa & Schiex, 2003; de Givry et al., 2003, 2005). It is a state-of-the-art Weighted CSP solver.

- PUEBLO 1.5 (Sheini & Sakallah, 2006). It is a pseudo-boolean solver. It ranked first on several categories of the 2005 Pseudo Boolean Evaluation.

- MINISAT+ (Eén & Sörensson, 2006). It is a pseudo-boolean solver that translates the problems into SAT and solves them with MiniSat. It ranked first on several categories of the 2005 Pseudo Boolean Evaluation.

Those instances taken from the pseudo-boolean evaluation were given in their original format to PUEBLO and MINISAT+. All other instances were translated from Max-SAT to PBO by partitioning the set of clauses into three sets: $\mathcal{H}$ contains the mandatory clauses $(C,\top)$, $\mathcal{W}$ contains the non-unary weighted clauses $(C,u<\top)$ and $\mathcal{U}$ contains the unary weighted clauses (l,u). For each hard clause $(C_j,\top)\in\mathcal{H}$ there is a pseudo boolean constraint $C'_j\geq 1$, where C'_j is obtained from C_j by replacing $\vee$ by $+$ and negated variables $\bar{x}$ by $1-x$. For each non-unary weighted clause $(C_j,u_j)\in\mathcal{W}$ there is a pseudo boolean constraint $C'_j+r_j\geq 1$, where C'_j is computed as before, and r_j is a new variable that, when set to 1, trivially satisfies the constraint. Finally, the objective function to minimize is,

$$\sum_{(C_j,u_j)\in\mathcal{W}} u_j r_j + \sum_{(l_j,u_j)\in\mathcal{U}} u_j l_j$$

8.3 Experimental Results

We divide the experiments in two parts. The purpose of the first part is to evaluate the impact of the different techniques of MINIMAXSAT and set the different parameters. Since some of the techniques can be effective in some benchmarks and useless or even counterproductive in some others (Brglez, Li, & Stallman, 2002), we aimed at finding a configuration such that MINIMAXSAT

8. http://www.cril.univ-artois.fr/PB06/

performs reasonably well on all the instances. The purpose of the second part is to compare MINIMAXSAT with alternative solvers. Since some of these solvers are specifically designed for some type of problems, we do not expect that MINIMAXSAT will outperform them. We rather want to show the robustness of MINIMAXSAT by showing that it is usually close in performance with the best alternative for each type of problems.

Results are presented in plots and tables. Regarding tables, the first column contains the name of the set of problems. The second column shows the number of instances. The remaining columns report the performance of the different solvers. Each cell contains the average *cpu* time that the solver required to solve all instances. If some solver could not solve all the instances of a set, a number inside brackets indicates the number of solved instances and the average *cpu* time only takes into account solved instances. If a cell contains a dash, it means that no instance could be solved within the time limit. Regarding plots, note that the legend goes in accordance with the performance of the solvers. The time limit was set to 900 seconds for each instance.

Our solver, written in C++, was implemented on top of MINISAT+ (Eén & Sörensson, 2006). Executions were made on a 3.2 Ghz Xeon computer with Linux. In all the experiments with random instances, samples had 30 instances and plots report mean *cpu* time in seconds.

8.4 Setting the Parameters of MINIMAXSAT

In the following we evaluate in order the importance of the following techniques inside MINIMAXSAT: lower bounding, soft probing, branching heuristics, learning and backjumping.

Starting from a basic version that guides search with the Jeroslow branching heuristic and has the rest of techniques deactivated, we analyze them one by one. Each analysis studies one technique and incorporates all the previously analyzed ones with the corresponding tuned parameters. In the three first experiments we only consider little but challenging instances generated randomly in which lower bounding plays a fundamental role to solve them. Finally, we consider structured instances in which learning and backjumping is required to solve them.

8.4.1 LOWER BOUNDING

In this experiment we analyze the impact of resolution-based lower bounding versus substraction-based lower bounding, as well as combined strategies. We considered the following combination of the two techniques: when SUP detects an inconsistency and the refutation tree is computed, we look at the resolvent with maximum size. If its size is less than or equal to a parameter K, then resolution-based lower bounding is applied, otherwise substraction-based lower bounding is applied. We tested $K = \{0,1,2,3,4,5,\infty\}$. Note that $K = 0$ corresponds to pure substraction-based lower bounding (and therefore is similar to the approach of Li et al., 2005), while $K = \infty$ corresponds to a pure resolution-based lower bounding.

The results are presented in Figure 3. As can be seen, the pure substraction-based lower bounding $K = 0$ is always the worst option. Better results are obtained as K increases. However, the improvement stops (or nearly stops) when $K = 3$. When $K > 3$ no significant improvement is noticed. The plot omits the $K = 4$ and $K = 5$ case for clarity reasons. Since higher values of K may produce new clauses of higher size and this may cause overhead in some instances, we set $K = 3$ for the rest of the experiments.

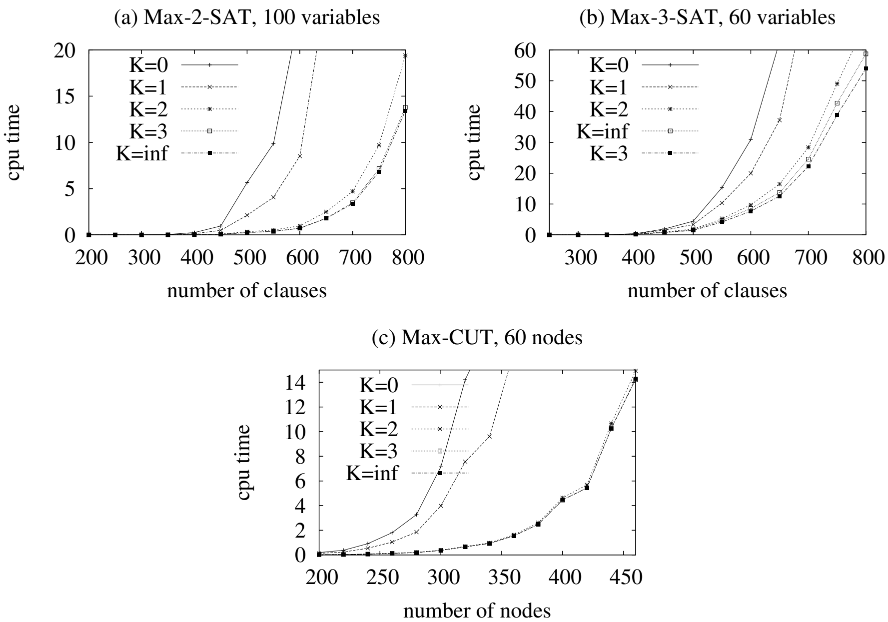


Figure 3: Performance of MINIMAXSAT with different mixed lower boundings ($K = 0, 1, 2, 3, \inf$).

8.4.2 SOFT PROBING

In our second experiment, we evaluate the impact of soft probing. In our preliminary experiments, we observed that soft probing was too time consuming, so we decided to limit soft probing as follows. Initially, we assign a *propagation level* of 0 to the variable to probe. Then, each new literal to propagate is assigned a *propagation level* $L+1$ if the literal that produces its propagation has level L. We limited probing to propagate literals with a maximum propagation level of M. We finally restricted $M \leq 2$ since it gives the best results. Note that a *propagation level* is not the same as a *decision level*.

We compare three alternatives: probing at each node of the search (S), probing as a pre-process before search (P) and no probing at all (N). The results, in Figure 4, indicates that probing during search is the worst option for Max-2-SAT and Max-3-SAT while it produces some improvement in Max-CUT. Finally, probing as a preprocessing gives slightly improvement for Max-2-SAT and the best results for Max-CUT. Note that soft probing as a preprocessing on Max-3-SAT has no effect and is omitted from the plot (its results are similar to N). Given these results, we decided to include soft probing only as a preprocessing.

8.4.3 JEROSLOW BRANCHING HEURISTIC

In the following experiment, we evaluate the importance of the weighted Jeroslow heuristic. Figure 5 shows the time difference between MINIMAXSAT with the Jeroslow heuristic as in the previous two experiments (Jeroslow) and without heuristic (None). The results indicates that guiding search with the Jeroslow heuristic gives important speed ups. Hence, we maintain the Jeroslow heuristic for MINIMAXSAT.

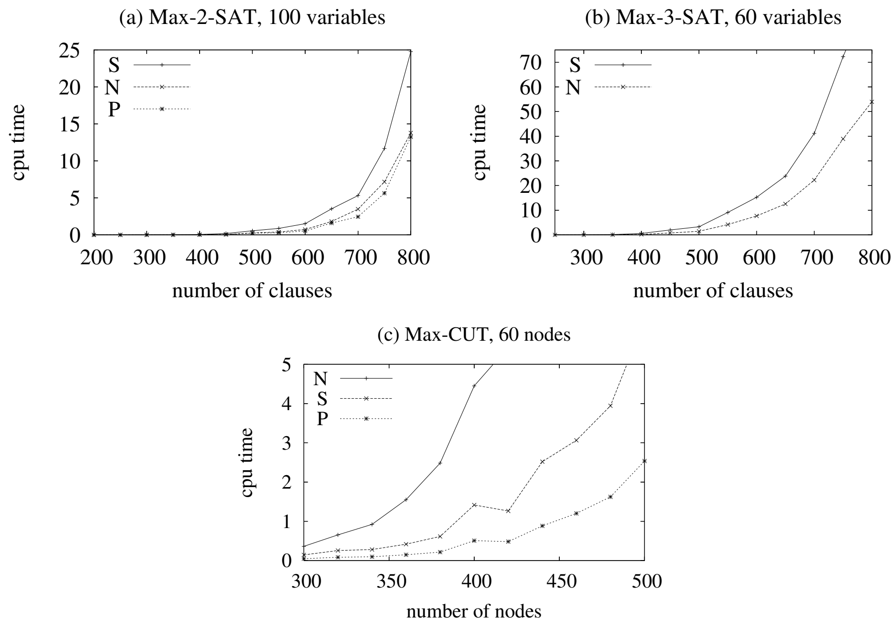


Figure 4: Performance of MINIMAXSAT without soft probing, with probing as preprocessing (P) and with probing during the search (S).

8.4.4 LEARNING, BACKJUMPING AND VSIDS

In the final experiment, we evaluate the importance of learning and backjumping. For these experiments we use structured instances, since it is well known that learning and backjumping are only useful in this type of problems. Besides, we also evaluate the importance of the VSIDS heuristic in combination with learning and backjumping. Recall that this heuristic was specially designed to work in cooperation with learning, so it is meaningless to analyze its effect by itself.

Table 6 reports the results of this experiment. The third column reports results without learning and backjumping but with the lower bounding, probing and the Jeroslow heuristic (None). The fourth column reports results adding learning and backjumping to the previous version (Learning). The fifth column reports results adding learning, backjumping but changing the Jeroslow heuristic by the VSIDS heuristic (VSIDS). The results show that MINIMAXSAT without learning and backjumping (None) is clearly the worst option. Significant improvements are obtained when learning and backjumping (Learning) are added. Finally, adding the VSIDS heuristic (VSIDS) improve further the results specially on the routing instances. Based on those results, we incorporated learning and backjumping to MINIMAXSAT.

Regarding the branching heuristic, for problems in which literals appear in hard clauses with both polarities it applies the VSIDS heuristic, otherwise the Jeroslow heuristic is computed in the root of the search tree as stated in Section 7. This choice is done once and for all before starting the search.

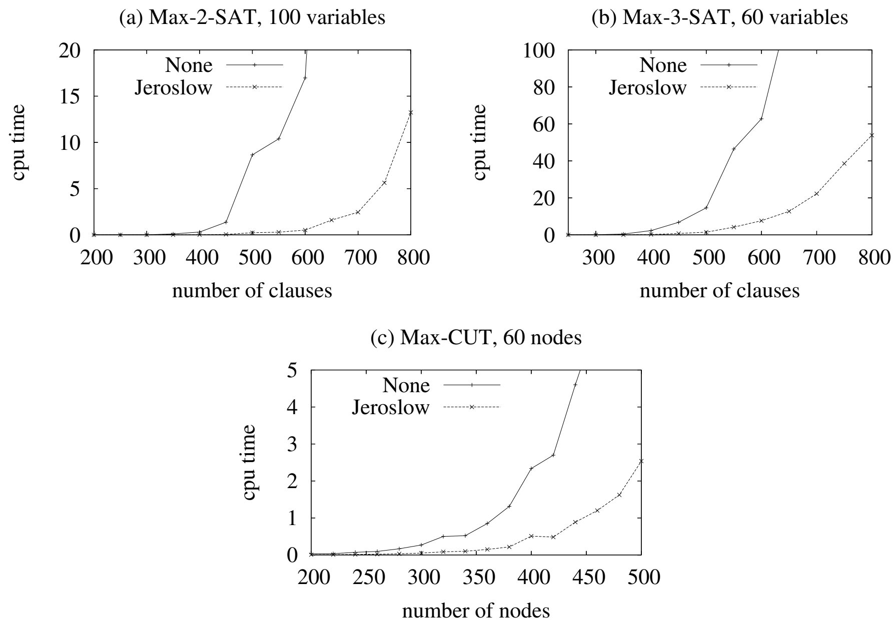


Figure 5: Performance of MINIMAXSAT without Heuristic (None) and with the Jeroslow heuristic computed in the root node of the search tree (Jeroslow).

Problem	n. inst.	None	Learning	VSIDS
Max-One 3col	40	–	29.06	15.41
Max-One cnt	3	13.57(1)	119.53	6.58
Max-One dp	6	16.11(4)	40.03	28.63
Max-One ezfact32	10	654.94(2)	0.70	0.77
Routing S3	5	22.26(4)	1.02	0.10
Routing S4	10	–	410.61(2)	91.09(9)

Figure 6: Structured instances.

8.5 Comparison with Other Boolean Optimizers

When reporting results, we will omit a solver if it cannot deal with the corresponding instances for technical reasons (e.g. it cannot deal with weighted clauses) or it performs extremely bad in comparison with the others.

Figure 7 contains plots with the results on different benchmarks. Plots *a* and *b* reports results on random unweighted Max-SAT instances. PUEBLO and MINISAT+ are orders of magnitude slower, so they are not included in the graphics. On Max-2-SAT (plot *a*), MINIMAXSAT lays between MAX-DPLL and MAXSATZ, which is the best option. On Max-3-SAT (plot *b*) MINIMAXSAT clearly outperforms MAX-DPLL and is very close to MAXSATZ, which is again the best. In both Max-2-SAT and Max-3-SAT MAXSATZ is no more than 3 times faster than MINIMAXSAT.

Plot c reports results on random Max-CUT instances. MINIMAXSAT performs slightly better than MAXSATZ, which is the second alternative.

On random Max-One (plot d) MINIMAXSAT is the best solver by far. Almost all instances are solved instantly while PUEBLO and MAX-DPLL require up to 10 seconds in the most difficult instances. MINISAT+ performs very poorly. The results on structured Max-One instances are reported in Figure 9. MINISAT+ seems to be the fastest in general. MINIMAXSAT is close in performance to PUEBLO. Note, however, that in the dp instances, MINIMAXSAT is the only system solving all instances.

Plot e reports the results on Random Max-Clique instances. MINIMAXSAT is the best solver, up to an order of magnitude faster than MAX-DPLL, the second best option. PUEBLO and MINISAT+ perform poorly again. Regarding the structured Dimacs instances, MINIMAXSAT is again the best option. It solves 36 instances within the time limit, while MAX-DPLL,MINISAT+ and PUEBLO solve 34, 22 and 18 respectively.

Plots f, g and h present the results on Combinatorial Auctions following different distributions. On the paths distribution, MINIMAXSAT is the best solver, twice faster than MAX-DPLL, which ranks second. On the regions distribution, MINIMAXSAT is the best solver while MAX-DPLL is the second best solver requiring double time. On the paths and regions distributions, PUEBLO and MINISAT+ perform very poorly. On the scheduling distribution, MINISAT+ is the best solver while MINIMAXSAT and MAX-DPLL are about one order of magnitude slower.

Results regarding the unsatisfiable DIMACS instances are presented in Figure 8. Note that all these instances have optimum cost 1. Hence, as soon as MINIMAXSAT find a solution of cost 1, all the clauses are declared hard and learning and backjumping can be applied when hard conflicts arise. The results indicate that MAXSATZ and MAX-DPLL do not solve any instance on some sets (Pret150 and Aim200), while MINIMAXSAT solves all sets of instances with the best times in all of them, except for the hole instances in which MAXSATZ is slightly faster. If we encode these problems in the most advantageous way for PUEBLO and MINISAT+, that is, as decision problems rather than optimization problems they solve all the instances with similar times to MINIMAXSAT.

On the planning instances (Fig. 10) PUEBLO is the best solver. MINIMAXSAT is the second best solver, TOOLBAR is the third and the last one is MINISAT+. This is not surprising since TOOLBAR does not perform learning over the hard constraints. Results regarding graph coloring instances are presented in Fig. 10. As can be observed, MINIMAXSAT is able to solve one more instance than TOOLBAR, while PUEBLO and MINISAT+ solve many less instances. On the Max-CSP problems (Fig. 10) TOOLBAR solves all the instances instantly while PUEBLO is the worst option unable to solve a lot of instances. MINIMAXSAT is clearly the second best solver and MINISAT+ is the third best performing solver. Note that both of them solve all the instances.

Results regarding the instances taken from the pseudo-boolean evaluation can be found in Figure 11. Note that this is the first time that a Max-SAT solver is tested on pseudo-boolean instances. Results indicate that no solver consistently outperforms the other and that MINIMAXSAT is fairly competitive with PUEBLO and MINISAT+.

¿From all these results we can conclude that MINIMAXSAT is a very robust Weighted Max-SAT solver. It is very competitive for pure optimization problems and for problems with lots of hard clauses and, sometimes, it is the best option.

As a final remark, note that MINIMAXSAT and almost all the previous benchmarks were submitted to the *Second Max-SAT Evaluation 2007*, a co-located event of the *Tenth International Conference on Theory and Applications of Satisfiability Testing*. Hence, the interested reader can find a

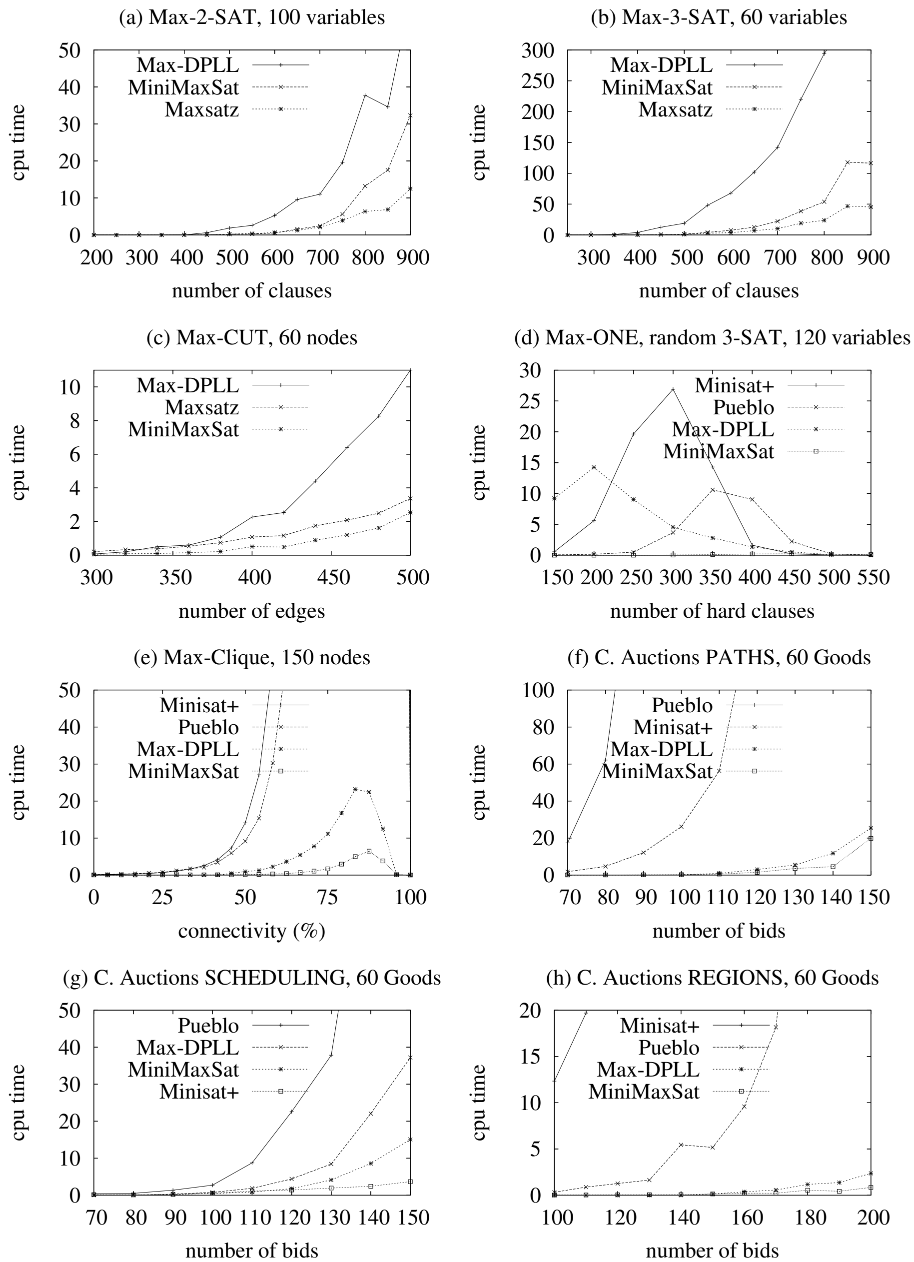


Figure 7: Plots of different benchmarks. Note that the order in the legend goes in accordance with the performance of the solvers.

more exhaustive comparison, including more instances and solvers, in the Second Max-SAT Evaluation 2007 web page[9]. The results of such evaluation showed that MINIMAXSAT was the best performing solver in two of the four existing categories.

9. http://www.maxsat07.udl.es/

Problem	n. inst.	MINIMAXSAT	MAXSATZ	MAX-DPLL
Dubois	13	0.02	148.18(7)	174.33(6)
Pret60	4	0.07	10.06	22.00
Pret150	4	0.01	–	–
Hole	5	8.68	8.34	28.00
Aim50	8	0.00	0.01	0.00
Aim100	8	0.00	9.55	172.00
Aim200	8	0.00	–	–

Figure 8: Unsatisfiable DIMACS instances.

Problem	n. inst.	MINIMAXSAT	PUEBLO	MINISAT+
3col80	10	0.15	0.10	0.02
3col100	10	2.25	1.73	0.12
3col120	10	20.49	14.52	0.74
3col140	10	38.33	83.17	1.61
cnt	3	6.59	0.13	0.12
dp	6	28.81	1.19(3)	1.21(4)
ezfact32	10	0.77	0.34	0.33

Figure 9: Structured Max-one instances.

Problem	n. inst.	Toolbar	MINIMAXSAT	PUEBLO	MINISAT+
Planning	71	4.02	3.81	0.16	7.40
Graph Coloring	22	49.29(16)	4.16(17)	68.50(11)	0.57(11)
Max-CSP DL	20	0.08	0.20	349.08(13)	8.60
Max-CSP DT	20	0.00	0.01	–	2.40
Max-CSP SL	20	0.01	0.03	123.67	0.48
Max-CSP ST	20	0.00	0.01	–	1.29

Figure 10: Results for WCSP and Max-CSP instances.

9. Related Work

Some previous work has been done about incorporating SAT-techniques inside a Max-SAT solver. Alsinet et al. (2005) presented a lazy data structure to detect when clauses become unit, but it requires a static branching heuristic. Argelich and Manyà (2006a) test different versions of a branch and bound procedure. One of these versions uses the two-watched literals, but it uses a very basic lower bounding. We can conclude that none of these previous approaches is as general as our use of the two-watched literals. As far as we know, the rest of Max-SAT solvers are based on *adjacency lists*. Therefore, they are presumably inefficient for unit propagation (Lynce & Silva, 2005), par-

Problem	n. inst.	MINIMAXSAT	PUEBLO	MINISAT+
misc	7	3.08(5)	8.51(5)	0.14(5)
Logic synthesis	17	82.55(2)	36.21(5)	253.93(5)
MPI	148	37.35(107)	32.04(101)	3.06(105)
MPS	16	22.65(5)	36.90(8)	8.50(8)
Routing	15	58.74(14)	5.96	13.09

Figure 11: Results for pseudo-boolean instances.

ticularly in the presence of long clauses. Argelich and Manyà (2006b) enhance a Max-SAT branch and bound procedure with learning over hard constraints, but it is used in combination with simple lower bounding techniques. An improved version is presented by Argelich and Manya (2007) with a more powerful lower bound, but it does not incorporate the two-watched literal scheme, backjumping, etc. To the best of our knowledge, no Max-SAT solver incorporates backjumping. Note that MINIMAXSAT restricts backjumping to the occurrence of hard conflicts. Related works on the integration of backjumping techniques into branch and bound include work by Zivan and Meisels (2007) for *Weighted CSP*, Manquinho and Silva (2004) for pseudo-boolean optimization, and Nieuwenhuis and Oliveras (2006) for *SAT Modulo Theories*.

Most Max-SAT solvers use variations of what we call substraction-based lower bounding. In most cases, they search for special patterns of mutually inconsistent subsets of clauses (Shen & Zhang, 2004; Xing & Zhang, 2005; Alsinet et al., 2005). For efficiency reasons, these patterns are always restricted to small sets of small arity clauses (2 or 3 clauses or arity less than 3). MINIMAXSAT uses a natural weighted extension of the approach proposed by Li et al. (2005). It was the first one able to detect inconsistencies in arbitrarily large sets of arbitrarily large clauses.

The idea of what we call resolution-based lower bounding was inspired from the WCSP domain (Larrosa, 2002; Larrosa & Schiex, 2003; de Givry et al., 2003, 2005) and it was first proposed in the Max-SAT context by Larrosa and Heras (2005) and further developed by Li et al. (2007), Heras and Larrosa (2006), and Larrosa et al. (2007). In these works, only special patterns of fixed-size resolution trees were executed. The use of simulated unit propagation allows MINIMAXSAT to identify arbitrarily large resolution trees. In the following example, we present two inconsistent subsets of clauses that are detected by MINIMAXSAT and transformed into an equivalent formula while previous solvers cannot transform them since they are limited to specific patterns:

- $\{(x_1,w_1),(x_2,w_2),(x_3,w_3),(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3,w_4)\}$
- $\{(x_1,w_1),(\bar{x}_1 \vee x_2,w_2),(\bar{x}_1 \vee \bar{x}_2 \vee x_3,w_3),(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4,w_4),(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee \bar{x}_4,w_5)\}$

In the first case, MINIMAXSAT replaces the clauses by $(\square,m)$ with $m = \min\{w_1,w_2,w_3,w_4\}$ and a set of compensation clauses. For the second case, MINIMAXSAT replaces it by $(\square,m)$ with $m = \min\{w_1,w_2,w_3,w_4,w_5\}$ and a set of compensation clauses. In both cases, the equivalence is preserved. However, other solvers in the literature detect those inconsistent subset of clauses but cannot transform the problem into an equivalent one (Li et al., 2007) or simply cannot detect them (Heras & Larrosa, 2006).

Our probing method to derive weighted unit clauses is related to the $2-RES$ and cycle rule of Heras and Larrosa (2006) and Larrosa et al. (2007), to failed literals of Li et al. (2006), and to singleton consistency in CSP (Debruyne & Bessière, 1999). Again, the use of simulated unit propagation allows MINIMAXSAT to identify arbitrarily large resolution trees.

10. Conclusions and Future Work

MINIMAXSAT is an efficient and very robust Max-SAT solver that can deal with hard and soft clauses as well as pseudo-boolean functions. It incorporates the best available techniques for each type of problems, so its performance is similar to the best specialized solver. Besides the development of MINIMAXSAT combining, for the first time, known techniques from different fields, the main original contribution of this paper is a novel lower bounding technique based on resolution.

MINIMAXSAT lower bounding combines in a very clean and elegant way most of the approaches that have been proposed in the last years, mainly based on unit-propagation-based lower bounding and resolution-based problem transformation. In this paper we use the information provided by the propagation queue (i) to determine a subset of inconsistent clauses and (ii) to determine a simple ordering in which resolution can be applied to increase the lower bound and generate an equivalent formula. However, this is not necessarily the best ordering to do so. It is easy to see that different orderings may generate resolvents and compensation clauses of different arities. If one selects the ordering that generates the smallest resolvents and compensation clauses the resulting formula may be presumably simpler. Future work concerns the study of such orderings, the development of VSIDS-like heuristics for soft clauses and backjumping techniques for soft conflicts.

Acknowledgments

We would like to thank to Niklas Eén and Niklas Sörensson for making MINISAT+ code publicly available. We are also grateful to the anonymous referees for their helpful suggestions on improving the paper.

This work has been partially supported by the Spanish Ministry of Education and Science through the projects TIN2006-15387-C03-02 (Heras and Larrosa) and TIN2004-03382 (Oliveras).

References

Alsinet, T., Manyà, F., & Planes, J. (2005). Improved Exact Solvers for Weighted Max-SAT. In *Proceedings of SAT'05*, Vol. 3569 of *LNCS*, pp. 371–377. Springer.

Argelich, J., & Manyà, F. (2006a). Exact Max-SAT solvers for over-constrained problems. *J. Heuristics*, *12*(4-5), 375–392.

Argelich, J., & Manyà, F. (2006b). Learning Hard Constraints in Max-SAT. In *Proceedings of CSCLP'06*, Vol. 4651 of *LNCS*, pp. 1–12. Springer.

Argelich, J., & Manya, F. (2007). Partial Max-SAT Solvers with Clause Learning. In *Proceedings of SAT'07*, Vol. 4501 of *LNCS*, pp. 28–40. Springer.

Barth, P. (1995). A Davis-Putnam Based Enumeration Algorithm for Linear pseudo-Boolean Optimization. Research report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany.

Brglez, F., Li, X., & Stallman, M. (2002). The role of a skeptic agent in testing and benchmarking of SAT algorithms. In *In Proceedings of SAT'02*, pp. 354–361.

Buro, M., & Büning, H. K. (1993). Report on a SAT Competition. *Bulletin of the European Association for Theoretical Computer Science*, *49*, 143–151.

Cha, B., Iwama, K., Kambayashi, Y., & Miyazaki, S. (1997). Local search algorithms for partial MAXSAT. In *Proceedings of AAAI'97*, pp. 263–268. The MIT Press.

Cooper, M., Cussat-Blanc, S., de Roquemaurel, M., & Régnier, P. (2006). Soft Arc Consistency Applied to Optimal Planning. In *Proceedings of CP'06*, Vol. 4204 of *LNCS*, pp. 680–684. Springer.

Davis, M., Logemann, G., & Loveland, G. (1962). A machine program for theorem proving. *Communications of the ACM*, *5*, 394–397.

de Givry, S., Heras, F., Larrosa, J., & Zytnicki, M. (2005). Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proceedings of the 19^{th} IJCAI*, pp. 84–89. Professional Book Center.

de Givry, S., Larrosa, J., Meseguer, P., & Schiex, T. (2003). Solving Max-SAT as weighted CSP. In *Proceedings of CP'03*, Vol. 2833 of *LNCS*, pp. 363–376. Springer.

Debruyne, R., & Bessière, C. (1999). Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of ICJAI'97*, pp. 412–417. Morgan Kaufmann.

Eén, N., & Sörensson, N. (2003). An Extensible SAT-solver. In *Proceedings of SAT'03*, Vol. 2919 of *LNCS*, pp. 502–518. Springer.

Eén, N., & Sörensson, N. (2006). Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, *2*, 1–26.

Fahle, T. (2002). Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA'02*, Vol. 2461 of *LNCS*, pp. 485–498. Springer.

Freeman, J. W. (1995). *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. thesis, University of Pennsylvania.

Fu, Z., & Malik, S. (2006). On Solving the Partial MAX-SAT Problem. In *Proceedings of SAT'06*, Vol. 4121 of *LNCS*, pp. 252–265. Springer.

Heras, F., & Larrosa, J. (2006). New Inference Rules for Efficient Max-SAT Solving. In *Proceedings of the 21^{th}AAAI*. AAAI Press.

Jeroslow, R. G., & Wang, J. (1990). Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, *1*, 167–187.

K. Leyton-Brown, M. P., & Shoham, Y. (2000). Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of ACM Conference on Electronic Commerce'00*, pp. 66–76.

Karloff, H. J., & Zwick, U. (1997). A 7/8-Approximation Algorithm for MAX 3SAT?. In *FOCS*, pp. 406–415.

Larrosa, J., & Heras, F. (2005). Resolution in Max-SAT and its relation to local consistency for weighted CSPs. In *Proceedings of IJCAI'05*, pp. 193–198. Professional Book Center.

Larrosa, J., Heras, F., & de Givry, S. (2007). A logical approach to efficient max-sat solving. In *Artificial Intelligence*. To appear.

Larrosa, J., & Schiex, T. (2003). In the quest of the best form of local consistency for weighted CSP. In *Proceedings of the 18th IJCAI*, pp. 239–244.

Larrosa, J. (2002). Node and Arc Consistency in Weighted CSP. In *Proceedings of AAAI'02*, pp. 48–53. AAAI Press.

Le Berre, D. (2001). Exploiting the real power of Unit Propagation Lookahead. In *Proceedings of LICS Workshop on Theory and Applications of Satisfiability Testing*.

Le Berre, D. (2006). The SAT4j project for Max-SAT.. http://www.sat4j.org/.

Li, C., Manyà, F., & Planes, J. (2005). Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers. In *Proceedings of CP'05*, Vol. 3709 of *LNCS*, pp. 403–414.

Li, C., Manyà, F., & Planes, J. (2007). New Inference Rules for Max-SAT. In *Journal of Artificial Intelligence Research*. To appear.

Li, C.-M., Manyà, F., & Planes, J. (2006). Detecting Disjoint Inconsistent Subformulas for Computing Lower Bounds for Max-SAT. In *Proceedings of the 21thAAAI*. AAAI Press.

Lynce, I., & Silva, J. P. M. (2003). Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *Proceedings of ICTAI'03*, pp. 105–111. IEEE Computer Society.

Lynce, I., & Silva, J. P. M. (2005). Efficient data structures for backtrack search SAT solvers. *Ann. Math. Artif. Intell.*, *43*(1), 137–152.

Manquinho, V. M., & Silva, J. P. M. (2004). Satisfiability-Based Algorithms for Boolean Optimization. *Ann. Math. Artif. Intell.*, *40*(3-4), 353–372.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pp. 530–535. ACM.

Nieuwenhuis, R., & Oliveras, A. (2006). On SAT Modulo Theories and Optimization Problems. In *Proceedings of SAT'06*, Vol. 4121 of *LNCS*, pp. 156–169. Springer.

Papadimitriou, C. (1994). *Computational Complexity*. Addison-Wesley, USA.

Sandholm, T. (1999). An Algorithm for Optimal Winner Determination in Combinatorial Auctions. In *Proceedings of IJCAI'99*, pp. 542–547. Morgan Kaufmann.

Sheini, H. M., & Sakallah, K. A. (2006). Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, *2*, 165–189.

Shen, H., & Zhang, H. (2004). Study of lower bounds for Max-2-SAT. In *Proceedings of AAAI'04*, pp. 185–190. AAAI Press / The MIT Press.

Silva, J. P. M., & Sakallah, K. A. (1996). GRASP - a new search algorithm for satisfiability. In *ICCAD*, pp. 220–227.

Smyth, K., Hoos, H. H., & Stützle, T. (2003). Iterated Robust Tabu Search for MAX-SAT. In *Proceedings of AI'03*, Vol. 2671 of *LNCS*, pp. 129–144. Springer.

Tompkins, D. A. D., & Hoos, H. H. (2004). UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT & MAX-SAT. In *Proceedings of SAT'04*, Vol. 3542 of *LNCS*, pp. 306–320. Springer.

Walsh, T. (2000). SAT v CSP. In *Proceedings of CP'00*, Vol. 1894 of *LNCS*, pp. 441–456. Springer.

Xing, Z., & Zhang, W. (2005). MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, *164*(1-2), 47–80.

Zhang, L., Madigan, C. F., Moskewicz, M. W., & Malik, S. (2001). Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proceedings of ICCAD'01*, pp. 279–285.

Zivan, R., & Meisels, A. (2007). Conflict directed Backjumping for MaxCSPs. In *Proceedings of IJCAI'07*, pp. 198–204.