MiniMaxSat: A New Weighted Max-SAT Solver

Federico Heras, Javier Larrosa, and Albert Oliveras

Universitat Politecnica de Catalunya,
Jordi Girona 1-3, 08034 Barcelona, Spain

Abstract. In this paper we introduce MINIMAXSAT, a new Max-SAT solver that incorporates the best 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 subtraction-based lower bounding; and lazy propagation with the two-watched literals scheme. Our empirical evaluation on a wide set of optimization benchmarks indicates that its performance 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*, *electronic markets*, *etc...*

There is a long tradition of theoretical work about the structural complexity [1] and approximability [2] 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 clauses). From a practical point of view, a significant progress has been made in the last 3 years [3,4,5,6,7,8]. As a result, there is a handful of new solvers that can deal, for the first time, with medium-sized instances.

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 in [9], 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, [7,8] are restricted to formulas in which all clauses are equally important, [3] is restricted to binary clauses, [5] seems to be efficient on very overconstrained problems (*i.e.*, only a small fraction of the clauses can be simultaneously satisfied), while [10] seems to be efficient on slightly overconstrained problems (*i.e.* almost all the clauses can be satisfied). The solver proposed in [11] is the only one that incorporates some learning, so it will presumably perform well on structured problems, but its lower bound computation is relatively weak, so it does not seem to be competitive in pure optimization problems.

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+ [12], so it borrows its capability to deal with pseudo-boolean problems and all the MiniSAT [13] 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 literals lazy propagation method. The main original contribution of MINIMAXSAT is that it implements a very efficient lower bounding technique. Specifically, it applies unit propagation in order to detect disjoint inconsistent clauses like in [8] and then it transforms the problem like in [4,14,5] to increment the lower bound. However, while in [4,14,5] 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, Section 3 overviews MINIMAXSAT, Sections 4 and 5 focus on its lower bounding and additional features, respectively. Section 6 reports experimental results and Section 7 presents related work. Finally, Section 8 concludes and points out possible future work.

2 Preliminaries

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. In the following, possibly subscripted capital letters A, B, C, D, and E will always represent clauses. 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 $var(C)$. 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*. 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 *weighted* clause is a pair (C, w), where C is a clause and w is the cost of its falsification, also called its *weight*. Many real problems contain 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 multiset 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, *Weighted* Max-SAT is the problem of finding a model of minimum cost. 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 formula $\mathcal{F}'$ is a *relaxation* of formula $\mathcal{F}$ (noted $\mathcal{F}' \sqsubseteq \mathcal{F}$) if they are defined over the same set of variables and the cost of any complete assignment in $\mathcal{F}'$ is less than or equal to the cost in $\mathcal{F}$ (non-models are considered to have cost infinity). We say that two formulas $\mathcal{F}'$ and $\mathcal{F}$ are *equivalent* (noted $\mathcal{F}' \equiv \mathcal{F}$) if $\mathcal{F}' \sqsubseteq \mathcal{F}$ and $\mathcal{F} \sqsubseteq \mathcal{F}'$.

If a formula contains clauses (C,u) and (C,v), they can be replaced by $(C,u+v)$ and if it contains a clause $(C,0)$, this may be removed. Both these transformation preserve equivalence. The empty clause may appear in a formula. If its weight is $\top$, it is clear that the formula does not have any model. If its weight is w, 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 4.

Mandatory clauses of size 1 (namely, $(l,\top)$) are called *facts*. When a formula contains a fact $(l,\top)$, 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. Note that most of them use a lazy implementation of UP based on the *two-watched literals scheme* [15].

3 Overview of MiniMaxSat

MiniMaxSat performs a *depth-first branch-and-bound* search on the tree of possible assignments, where internal nodes represent partial assignments and leaf nodes represent complete assignments. Each internal node has two children: the two possible extensions of its associated assignment with respect to one of the unassigned variables. At an arbitrary search point, the algorithm tries to detect a conflict, which means that the current partial assignment cannot be successfully extended. We distinguish two types of conflicts: *hard* conflicts indicate that there is no model extending the current partial assignment (namely, all the mandatory clauses cannot be satisfied), and *soft* conflicts indicate that the current partial assignment cannot be extended to an optimal assignment. Hard conflicts are detected when unit propagation leads to the empty mandatory clause $(\Box,\top)$. The detection of soft conflicts requires that the algorithm maintains two values during 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. Note 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.

An algorithmic description of MiniMaxSat is presented in *Algorithm 1*. The algorithm uses a propagation queue Q which contains all facts pending propagation. Once propagated, literals are not removed from Q, but rather marked as such. The algorithm also uses an array $V(l)$ which accumulates the weight of all soft clauses that have become unit over l (namely, clauses $(A \vee l, w)$ such that the current assignment falsifies A).

Before starting the search, a good initial upper bound is obtained with a local search method (line 1) which may yield the identification of some new hard clauses. In our current implementation we use Ubcsat [16] with default parameters. The selected local search algorithm is *IROTS* (*Iterated Robust Tabu Search*). Next, the queue Q is initialized with all the facts in the resulting formula (line 2). The main loop starts in line 3 and each iteration is in charge of propagating all pending facts (line 4) and, if no conflict is detected, attempting the extension of the current partial assignment (line 10). Pending facts in Q are propagated in function `Propagate` (line 4), which may return a hard or soft conflict (see next Section for details). If a hard conflict is encountered (line 5) the conflict is analyzed, a new hard clause is learnt and backjumping is performed. This is done as it is customary in classical SAT solvers such as Chaff [15]. If a soft conflict is encountered (line 6) chronological backtracking is performed. Note that this does not affect the overall completeness of the procedure, but some subtle implementation details are necessary. If no conflict is found (line 10), a literal is heuristically selected and added to Q for propagation in the next iteration. However, if the current assignment is complete (line 7), the upper bound is updated. Search stops if a zero-cost solution is found because it cannot be further improved (line 8). Else, chronological backtracking is performed (line 9). 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 2 describes function `Propagate` that performs unit propagation (`UP`) which propagates facts (line 18). It iterates over the non-propagated literals l in Q (line 11). Firstly, the cost of falsifying $\bar{l}$ (which is recorded in $V(\bar{l})$) is added to the lower bound (line 12). Secondly, if a hard clause becomes a fact (line 13), the corresponding literal is added to Q for future propagation (line 14). Finally, if a soft clause becomes unit (q, u) (line 16), its weight u is added to $V(q)$ (line 17). If during this process a hard conflict is detected, the function returns it (line 15). Else, the algorithm attempts to detect a soft conflict with a call to procedure `improveLB` (line 20, see Section 4 for details), and it returns the soft conflict if it is found (line 21). Finally, if no conflict is detected, the function returns *None* (line 22).

Note that `Propagate` only needs to identify when original (soft or hard) clauses have all their literals but one falsified. Thus, we use the *two-watched literals* scheme [15] in both hard and soft clauses. Note that any changes to lb or $V(l)$ must be restored upon backtracking.

4 Lower Bounding in MiniMaxSat

In the following, we consider an arbitrary search state of MiniMaxSat before the call to `improveLB`. Such a search state is uniquely characterized by the current assignment. The current assignment determines the *current subformula* which is the

Algorithm 1. MINIMAXSAT basic structure

```
Function Search() : integer
1   $ub$ := LocalSearch() ;
2   InitQueue($Q$) ;
3   Loop
4       Propagate() ;
5       if Hard Conflict then
            Analyze() ;
            if Top Level Hard Conflict then return $ub$ ;
            else
                LearnClause() ;
                Backjumping() ;
        else
6           if Soft Conflict then
                ChronologicalBactracking() ;
                if End of Search then return $ub$ ;
            else
7               if all variables assigned then
                    $ub := lb$ ;
8                   if $ub = 0$ then return $ub$ ;
9                   ChronologicalBactracking() ;
                    if End of Search then return $ub$ ;
10              else
                    $l$ := SelectLiteral() ;
                    Enqueue($Q, l$) ;
```

original formula *conditioned* by the current assignment. The current subformula has the lower bound as the weight of the empty clause $(\Box, lb)$. Similarly, value $V(l)$ defines unit clause $(l, V(l))$. Recall that such a unit clause is the aggregation of all the original clauses that have become unit over l due to the current partial assignment.

MINIMAXSAT improves its lower bound in procedure `improveLB` (called in line 20 of *Algorithm 2*). It does so by deriving new soft empty clauses $(\Box, w)$ through a weighted resolution process. Such clauses are added to the original $(\Box, lb)$ clause producing an increment of the lower bound. *Weighted resolution* (also called *Max-RES*) [4], is a rule that *replaces* two *clashing* clauses $(x \vee A, u)$ and $(\bar{x} \vee B, w)$ by the following set of clauses $\{(A \vee B, m), (x \vee A, u-m), (\bar{x} \vee B, w-m), (x \vee A \vee \bar{B}, m), (\bar{x} \vee \bar{A} \vee B, m)\}$, [1] where $m = \min\{u, w\}$ and hard clauses are treated as if their cost was infinity (*i.e.* $\top - u = \top$). The first clause is called the *resolvent* and the other clauses are called *compensation clauses*. The transformation preserves equivalence as defined in Section 2.

[1] When A is the empty clause, $\overline{A}$ represents a tautology.

Algorithm 2. Functions related with the search algorithm

```
Function UP() : conflict
    while (Q contains non-propagated literals) do
11      $l :=$ PickNonPropagatedLiteral($Q$); MarkAsPropagated($l$) ;
12      $lb := lb + V(\bar{l})$ ;
13      foreach Hard clause that has become unit over literal $q$ do
14          Enqueue($Q, q$) ;
15          if $\{\bar{q}\} \in Q$ then return Hard Conflict ;
16      foreach Soft clause with weight $u$ that has become unit over literal $q$ do
17          $V(q) = V(q) + u$ ;
    return None ;
Function Propagate() : conflict
18  $c :=$ UP() ;
19  if $c =$ Hard Conflict then return $c$ ;
20  improveLB() ;
21  if $lb \geq ub$ then return Soft Conflict ;
22  return None ;
```

The last two compensation clauses may lose the clausal form, so the following rule [5] may be needed to recover it:

$$CNF(A \vee \overline{\bar{l} \vee B}, u) = \begin{cases} A \vee \bar{l} & : \quad |B| = 0 \\ \{(A \vee \bar{l} \vee B, u)\} \cup CNF(A \vee \bar{B}, u) & : \quad |B| > 0 \end{cases}$$

Example 1. *If we apply weighted resolution to the following clauses* $\{(x \vee y, 3), (\bar{x} \vee y \vee z, 4)\}$ *we obtain* $\{(y \vee y \vee z, 3), (x \vee y, 3-3), (\bar{x} \vee y \vee z, 4-3), (x \vee y \vee \overline{(y \vee z)}, 3), (\bar{x} \vee \bar{y} \vee y \vee z, 3)\}$. *The first and fourth clauses 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. We apply CNF rule to the fourth clause to obtain two new clauses* $CNF(x \vee y \vee \overline{(y \vee z)}, 3) = \{(x \vee y \vee \bar{y} \vee z), 3), (x \vee y \vee \bar{z}, 3)\}$. *Note that the first new clause is a tautology. Therefore, we obtain the equivalent formula* $\{(y \vee z, 3), (\bar{x} \vee y \vee z, 1), (x \vee y \vee \bar{z}, 3)\}$.

As a first step, `improveLB` performs *unit neighborhood resolution* (UNR) [17,4], which only resolves on pairs of clashing unit clauses. It produces an immediate increment of the lower bound (*i.e.*, the weight of the empty clause) as it is illustrated in the following example,

Example 2. *Consider a search state with two unassigned variables x and y in which the lower bound is* $lb = 3$, $V(x) = V(y) = 1$, $V(\bar{x}) = V(\bar{y}) = 2$ *and a clause* $(x \vee y, 3)$. *This is equivalent to the formula* $\{(\square, 3), (x, 1), (y, 1), (\bar{x}, 2), (\bar{y}, 2), (x \vee y, 3)\}$. *UNR would resolve on clauses* $(x, 1)$ *and* $(\bar{x}, 2)$ *replacing them by* $(\bar{x}, 1)$ *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* $(y, 1)$ *and* $(\bar{y}, 2)$ *replacing them by* $(\bar{y}, 1)$ *and* $(\square, 1)$. *The two empty clauses can be grouped into* $(\square, 4+1=5)$. *So, the new equivalent formula is* $\{(\square, 5), (\bar{x}, 1), (\bar{y}, 1), (x \vee y, 3)\}$ *with a higher lower bound of* 5.

As a second step we execute a *simulation of unit propagation* (SUP) in which soft clauses are treated as if they were hard. As seen in the previous section, unit propagation uses a propagation queue Q. In the following, we assume that together with each literal l, the queue Q also contains its *reason*: the clause $(A \vee l, w)$ that cause its unit propagation. If SUP yields a conflict, it means that there is a subset of (soft or hard) clauses that cannot be simultaneously satisfied. Let m be the minimum weight among these clauses. It is easy to see that the extension of the current partial assignment to the unassigned variables will have a cost of at least m. Besides, such a cost can be made explicit by a sequence of resolution steps. A resolution tree is built from the propagation queue Q as follows: let C_0 be the conflicting clause. Traverse Q *from tail to head* 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 $\Box$. Once the resolution tree is computed, weighted resolution can actually be done with the actual soft clauses and, as a result, the empty clause $(\Box, m)$ will be derived. Finally, all clauses used in the process will be replaced by $(\Box, m)$ and the corresponding compensation clauses, thus obtaining an equivalent formula with a lower bound increment of m. It is important to remark that this transformation preserves equivalence since all clauses are used at most once in the resolution process but we have to undo the transformation upon backtracking. We call this procedure *resolution-based* lower bounding.

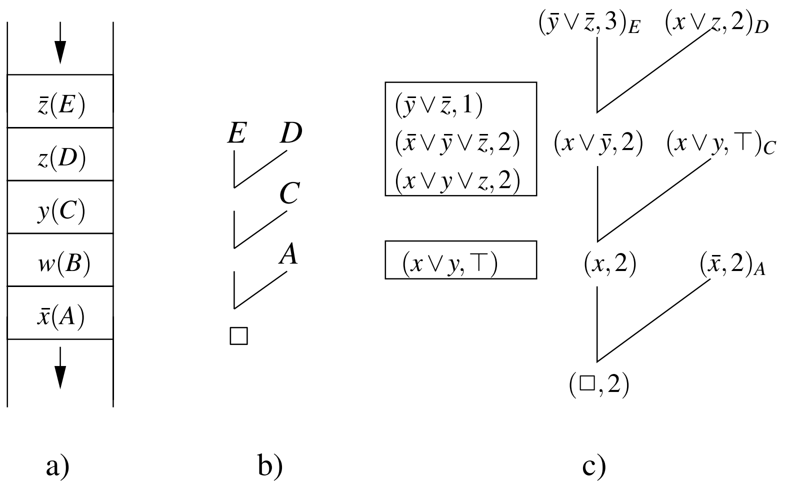
Example 3. *Consider formula* $\mathcal{F} = \{(\bar{x}, 2)_A, (x \vee w, 1)_B, (x \vee y, \top)_C, (x \vee z, 2)_D, (\bar{y} \vee \bar{z}, 3)_E\}$, *where each clause is identified by a subindex for future reference.*

Step 1. Apply SUP. *Initially, the unit clause A is enqueued producing* $Q = [\bar{x}(A)]$ *(within parenthesis, we indicate the reason of a literal). Then* $\bar{x}$ *is propagated. The resulting formula is* $\{(w, 1)_B, (y, \top)_C, (z, 2)_D, (\bar{y} \vee \bar{z}, 3)_E\}$ *and* Q *becomes* $[\bar{x}(A), w(B), y(C), z(D)]$. *Literal w is propagated. The resulting formula is* $\{(y, \top)_C, (z, 2)_D, (\bar{y} \vee \bar{z}, 3)_E\}$ *and no new unit clauses are generated. Literal y is propagated. The resulting formula is* $\{(z, 2)_D, (\bar{z}, 3)_E\}$ *and a new unit clause is enqueued producing* $Q = [\bar{x}(A), w(B), y(C), z(D), \bar{z}(E)]$. *Since z and* $\bar{z}$ *are inside Q, a conflict is detected and SUP stops. Note that E is the conflicting clause. Figure 1.a shows the state of Q after the propagation.*

Step 2. Build the resolution tree. *Starting from the tail of Q the first clause clashing with the conflicting clause E is D. Resolution between E and D generates the resolvent* $x \vee \bar{y}$. *The first clause clashing with it is C, producing resolvent x. The next clause clashing with it is A and resolution generates* $\Box$. *Figure 1.b shows the resulting resolution tree. The minimum weight among the involved clauses is* 2.

Step 3. Transform the problem. *We apply weighted resolution as indicated by the tree computed in Step 2. Figure 1.c graphically shows the result of the process. Leaf clauses are the original 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* $((\Box, 2))$ *and all compensation clauses. That is, the resulting formula is* $\mathcal{F}' = \{(x \vee w, 1), (x \vee y, \top), (\bar{y} \vee \bar{z}, 1), (\Box, 2), (x \vee y \vee z, 2), (\bar{x} \vee \bar{y} \vee \bar{z}, 2)\}$. *Note that* $\mathcal{F} \equiv \mathcal{F}'$.

$$\mathcal{F} = \{(\bar{x},2)_A, (x \vee w, 1)_B, (x \vee y, \top)_C, (x \vee z, 2)_D, (\bar{y} \vee \bar{z}, 3)_E\}$$



$\bar{z}(E)$
$z(D)$
$y(C)$
$w(B)$
$\bar{x}(A)$
E D
C
A
$\Box$
$(\bar{y} \vee \bar{z}, 1)$
$(\bar{x} \vee \bar{y} \vee \bar{z}, 2)$
$(x \vee y \vee z, 2)$
$(x \vee y, \top)$
$(\bar{y} \vee \bar{z}, 3)_E$ $(x \vee z, 2)_D$
$(x \vee \bar{y}, 2)$ $(x \vee y, \top)_C$
$(x, 2)$ $(\bar{x}, 2)_A$
$(\Box, 2)$
a)
b)
c)

$$\mathcal{F}' = \{(x \vee w, 1), (x \vee y, \top), (\bar{y} \vee \bar{z}, 1), (\Box, 2), (x \vee y \vee z, 2), (\bar{x} \vee \bar{y} \vee \bar{z}, 2)\}$$
$$\mathcal{F}'' = \{(x \vee w, 1), (x \vee y, \top), (\bar{y} \vee \bar{z}, 1), (\Box, 2)\}$$

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

An alternative to problem transformation through resolution is to identify the lower bound increment m and then subtract it from all the clauses that would have participated in the resolution tree. This procedure is reminiscent of the lower bound computed in [7] and we call it *subtraction-based* lower bounding.

Example 4. *Consider formula $\mathcal{F}$ from the previous example. Steps 1 and 2 are identical. However, subtraction-based lower bounding would replace Step 3 by Step 3' that subtracts weight 2 from the clauses that appear in the resolution tree and then adds $(\Box, 2)$ to the formula. The result is $\mathcal{F}'' = \{(x \vee w, 1), (x \vee y, \top), (\bar{y} \vee \bar{z}, 1), (\Box, 2)\}$. Note that $\mathcal{F}'' \sqsubseteq \mathcal{F}$, so its lower bound is also a lower bound of $\mathcal{F}$, but they are not equivalent. Hence, $\mathcal{F}''$ cannot be used in the subsequent search and if no soft conflict is immediately detected, this transformation has to be undone before continuing the search.*

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 find 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 is usually 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 as it would happen with the subtraction-based approach. In our experiments, we found that no scheme was systematically better than the other. We also found 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 only applies resolution-based lower bounding if the largest resolvent in the resolution tree has arity less than 4. Observe that compensation clauses will contain at most 4 literals. Otherwise, it applies subtraction-based lower bounding.

5 Additional Features of MINIMAXSAT

5.1 Probing

Probing is a well-known SAT technique that allows the formulation of hypothetical scenarios [18]. The idea is to temporarily assume that l is a fact and then execute unit propagation. If UP yields a conflict, we know that $\bar{l}$ is indeed a fact. 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 search, so it is normally done as a preprocess 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 facts, it may be used to make explicit weighted unit clauses. As in SAT, the idea is to temporarily assume that l is a fact and then *simulate* unit propagation (*i.e.*, execute SUP()). Then, we build the resolution tree T from the propagation queue Q. If all the clauses in T are hard, we know that $\bar{l}$ is indeed a fact. Else, we can reproduce T applying Max-RES with the actual clauses and derive a unit clause $(\bar{l}, m)$ where m is the minimum weight among the clauses in T. 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 2) an initial non-trivial lower bound of $min\{u, w\}$. We tested probing during search and as a preprocessing in several benchmarks. We observed empirically that probing as a preprocessing was the best option as it is in SAT.

Example 5. *Consider formula* $\mathcal{F} = \{(x \vee y, 1)_A, (x \vee z, 1)_B, (\bar{y} \vee \bar{z}, 1)_C\}$. *If we assume* $\bar{x}$ *by adding it to Q and then execute SUP a conflict is reached. We obtain* $Q = [\bar{x}(\emptyset), y(A), z(B), \bar{z}(C)]$ *and we detect that C is a conflicting clause. The clauses involved in the refutation are C, B, and A. Resolving clauses C and B results in* $\{(x \vee y, 1)_A, (x \vee \bar{y}, 1), (x \vee y \vee z, 1), (\bar{x} \vee \bar{y} \vee \bar{z}, 1)\}$. *The resolution of the previous resolvent and A produces the (equivalent) formula* $\mathcal{F}' = \{(x, 1), (x \vee y \vee z, 1), (\bar{x} \vee \bar{y} \vee \bar{z}, 1)\}$.

5.2 Branching Heuristic

For problems where all literals appear in hard clauses in only one polarity, a weighted version of the *Two-sided Jeroslow Wang* heuristic [14] is computed in the root node and used in the subsequent search (the importance of each clause is multiplied by its weight). For problems where literals appear in hard clauses with both polarities it applies the native VSIDS-like heuristic [15] of MiniSat. In both cases, if some literal l accomplishes $V(l) + lb \geq ub$ at some node of the search tree, then $\bar{l}$ is the selected literal to assign and l is never assigned.

5.3 Pseudo-boolean Optimization

MINIMAXSAT can solve *pseudo-boolean optimization problem* [19,12] of the form:
(1) minimize $\sum_{j=1}^{n} c_j \cdot x_j$
(2) 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. (1) is the *objective function* and (2) is the set of *pseudo-boolean constraints*. MINIMAXSAT uses MINISAT+ to transform pseudo-boolean constraints into hard clauses. That is, it determines heuristically the most appropriate encoding to hard clauses through *adders*, *sorters* or *BDDs*. Regarding the objective function, for each pair $c_j \cdot x_j$, a new soft unit clause $(\bar{x}_j, c_j)$ is added.

6 Experimental Results

We compare MINIMAXSAT with several optimizers from different communities:

- MAXSATZ [8,20]. Specialized unweighted Max-SAT solver. It applies a powerful subtraction lower bounding [8] plus limited transformation rules [20].
- MAX-DPLL [14,5] is a Weighted Max-SAT solver that performs a restricted form of resolution lower-bounding. MAX-DPLL is part of the TOOLBAR package.
- TOOLBAR [17,21,22,23]. It is a state-of-the-art Weighted CSP solver.
- PUEBLO 1.5 [19] is a pure pseudo-boolean solver.
- MINISAT+ [12] is a pseudo-boolean solver that translates pseudo-boolean problems into SAT and solves them with MiniSAT.

When reporting results, we will omit a solver if it cannot deal with the corresponding instances or it performs extremely bad. Results are presented in plots and tables. The first column of each table contains the name of the set of problems and the second shows the number of instances. The rest of columns report the performance of the solvers. Each cell contains the average CPU time that the solver required to solve all instances. If not all the instances were solved within the time limit (600 seconds), a number inside brackets indicates the number of solved instances and the average CPU time only takes into account solved instances. Note that in the plots the order of the legend goes in accordance with the performance of the solvers. All experiments were conducted on a 3.2 Ghz Pentium 4 computer with Linux.

The following benchmarks were considered:

- Random Max-2-SAT instances with 100 variables and clauses ranging from 200 to 900. Max-3-SAT instances with 80 variables and clauses ranging from 300 to 700. See [14].

- Random Max-CUT instances [14] with 60 nodes and the number of edges ranging from 300 to 500.
- Random and structured Max-Clique instances [5]. The random instances have 150 nodes and the edge density is ranged from 0 to 100 per cent. The structured instances correspond to the 66 instances of Dimacs Challenge.
- Combinatorial Auctions [5]. The instances were generated with CATS [24]. Three distributions were considered: *paths*, *scheduling* and *regions*. The number of goods is fixed to 60 and the number of bids is varied differently for each distribution.
- Max-One instances [5]. We have selected some SAT instances for which we have solved the Max-One problem. We considered structured instances coming from the 2002 SAT Competition [11] and random 3-SAT instances with 120 variables and ranging the number of clauses from 150 to 550.
- WCSP instances. Structured Planing instances [25] containing both boolean and non-boolean variables and hard and soft constraints. Random binary Max-CSP instances have three or four values per variable and only soft constraints. Depending on the number of constraints and the number of forbidden tuples, 4 distributions were generated: Dense Loose (DL), Dense Tight (DT), Sparse Loose (SL) and Sparse Tight (ST) [21]. WCSP instances were translated to Weighted Max-SAT using the direct encoding [26].
- Small integer optimization pseudo-boolean instances coming from the 2006 Pseudo-Boolean Evaluation. We considered some industrial instances corresponding to *logic synthesis* , and some handmade instances including *Misc* (*garden*), *min prime* and *MPS* (*miplib*).

Figure 2 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 Max-CUT instances. In these problems, MINIMAXSAT performs slightly better than MAXSATZ, which is the second alternative.

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 option. PUEBLO and MINISAT+ perform poorly again. Regarding the structured Dimacs instances, MINIMAXSAT is again the best option. It solves 34 instances within the time limit, while TOOLBAR,MINISAT+ and PUEBLO solve 29, 19 and 14 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, MAX-DPLL is the best solver while MINIMAXSAT 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 MAX-DPLL and MINIMAXSAT are about one order of magnitude slower.

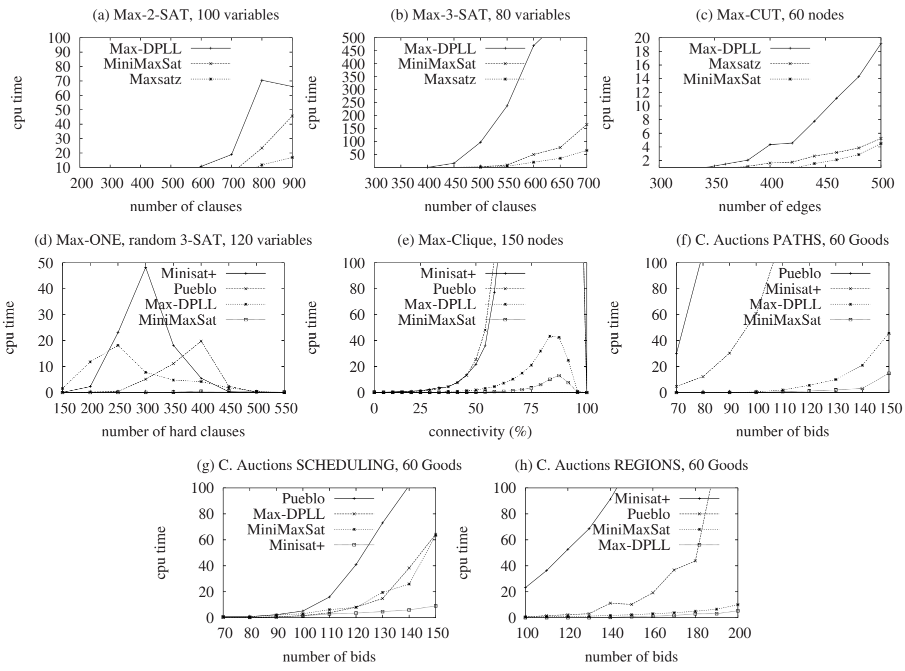


Fig. 2. Plots of different benchmarks. Note that the order in the legend goes in accordance with the performance of the solvers.

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 20 seconds in the most difficult instances. Minisat+ performs very poorly. The results on structured Max-One instances are reported in Figure 3. 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 system solving more instances.

On structured Planning WCSP instances (Fig. 4) 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. However, on pure optimization Max-CSP problems (Fig. 4) Toolbar solves all the instances instantly while Pueblo performs very poorly. MiniMaxSat is clearly the second best solver on DL instances, while Minisat+ is the second best option on DT and ST tight instances.

Results regarding pseudo-boolean instances can be found in Figure 5. 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+.

We can conclude that MiniMaxSat is the most 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.

Problem	n. inst.	MiniMaxSat	Pueblo	Minisat+
3col80	10	0.25	0.15	0.05
3col100	10	2.90	2.55	0.26
3col120	10	28.77	21.23	1.50
3col140	10	56.57	122.59	3.86
cnt	3	9.30	0.25	0.25
dp	6	11.75(5)	1.82(3)	2.40(4)
ezfact32	10	1.49	0.69	0.65

Fig. 3. Structured Max-one instances

Problem	n. inst.	Toolbar	MiniMaxSat	Pueblo	Minisat+
Planning	71	8.22	2.19	0.28	13.64
DL	20	0.14	2.20	302.85(8)	27.17
DT	20	0.00	7.48	0(0)	5.33
SL	20	0.01	33.08	83.30(18)	1.30
ST	20	0.00	18.04	0(0)	4.29

Fig. 4. Results for WCSP instances

Problem	n. inst.	MiniMaxSat	Pueblo	Minisat+
Garden	7	2.87(5)	13.60(5)	0.28(5)
Logic synthesis	17	26.33(2)	57.60(5)	4.21(2)
Min prime	156	20.94(111)	13.20(106)	7.58(112)
Miplib	17	34.50(5)	51.84(9)	21.48(9)

Fig. 5. Results for pseudo-boolean instances

7 Related Work

Some previous work has been done about incorporating SAT-techniques inside a Max-SAT solver. In [10] a lazy data structure to detect when clauses become unit is presented but it requires a static branching heuristic, so it is not as general as our extension of the two-watched literals. As far as we know, the rest of Max-SAT solvers are based on *adjacency lists* that are inefficient for unit propagation [27]. In [11] a Max-SAT branch and bound is powered with learning over hard constraints, but it is used in combination of simple lower bounding techniques. To the best of our knowledge, no Max-SAT solver incorporates backjumping. Note that MiniMaxSat restricts backjumping to the ocurrence of hard conflicts. Related frameworks that backjump after soft conflicts include [28] for *WCSP*, [29] for pseudo-boolean optimization and [30] for *SMT*.

Most Max-SAT solvers use what we call subtraction-based lower bounding. In most cases, they search for special patterns of mutually inconsistent subsets of clauses [3,6,10]. 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 in [7]. 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 [17,21,22,23] and it was first proposed in the Max-SAT context in [4] and further developed in [20,14,5]. 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.

Our probing method to derive weighted unit clauses is related to the $2-RES$ and cycle rule of [14,5] and to failed literals in [8]. Again, the use of simulated unit propagation allows MiniMaxSat to identify arbitrarily large resolution trees.

8 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 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 subsumes in a very clean an elegant way most of the approaches that have been proposed in the last years. Future work concerns the development of VSIDS-like heuristics for soft clauses, backjumping techniques for soft conflicts and the study of domain-specific branching heuristics.

References

1. Papadimitriou, C.: Computational Complexity. Addison-Wesley, USA (1994)
2. Karloff, H.J., Zwick, U.: A 7/8-Approximation Algorithm for MAX 3SAT. In: FOCS. (1997)
3. Shen, H., Zhang, H.: Study of lower bounds for Max-2-SAT. In: AAAI. (2004)
4. Larrosa, J., Heras, F.: Resolution in Max-SAT and its relation to local consistency for weighted CSPs. In: IJCAI. (2005)
5. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient max-sat solving. In: Available at the Computing Research Repository (http://arxiv.org/PS_cache/cs/ps/0611/0611025.ps.gz). (2006)
6. Xing, Z., Zhang, W.: MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. Artificial Intelligence **164** (2005) 47–80
7. Chu Min Li, F.M., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In: Proc. of the 11th CP, Sitges, Spain (2005)
8. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In: AAAI. (2006)
9. Cha, B., Iwama, K., Kambayashi, Y., Miyazaki, S.: Local search algorithms for partial MAXSAT. In: AAAI/IAAI. (1997) 263–268
10. Alsinet, T., Manya, F., Planes, J.: Improved exact solver for weighted max-sat. In SAT'05.
11. Argelich, J., Manyà, F.: Learning hard constraints in max-sat. In: CSCLP-2006. (2006) 1–12
12. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into sat. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 1–26
13. Eén, N., Sörensson, N.: An extensible sat-solver. In: Proceedings of SAT03. (2003) 502–518
14. Heras, F., Larrosa, J.: New inference rules for efficient max-sat solving. In: AAAI. (2006)
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: DAC. (2001) 530–535
16. Tompkins, D.A.D., Hoos, H.H.: Ubcsat: An implementation and experimentation environment for sls algorithms for sat & max-sat. In: SAT. (2004)
17. Larrosa, J.: Node and arc consistency in weighted CSP. In: AAAI. (2002) 48–53
18. Lynce, I., Silva, J.P.M.: Probing-based preprocessing techniques for propositional satisfiability. In: ICTAI. (2003) 105–
19. Sheini, H.M., Sakallah, K.A.: Pueblo: A hybrid pseudo-boolean sat solver. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 165–189
20. Li, C.M., Manyà, F., Planes, J.: New inference rules for max-sat. In: Submitted. (2006)
21. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: Proc. of the 18th IJCAI, Acapulco, Mexico (2003)

22. de Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving max-SAT as weighted CSP. In: Proc. of the 9th CP, Kinsale, Ireland, LNCS 2833. Springer Verlag (2003) 363–376
23. de Givry, S., Heras, F., Larrosa, J., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. of the 19th IJCAI, Edinburgh, U.K. (2005)
24. K. Leyton-Brown, M.P., Shoham, Y.: Towards a universal test suite for combinatorial auction algorithms. ACM E-Commerce (2000) 66–76
25. Cooper, M., Cussat-Blanc, S., de Roquemaurel, M., Régnier, P.: Soft arc consistency applied to optimal planning. In: CP. (2006) 680–684
26. Walsh, T.: SAT v CSP. In: CP. (2000) 441–456
27. Lynce, I., Silva, J.P.M.: Efficient data structures for backtrack search sat solvers. Ann. Math. Artif. Intell. **43** (2005) 137–152
28. Zivan, R., Meisels, A.: Conflict directed backjumping for maxcsps. In: IJCAI. (2007)
29. Manquinho, V.M., Silva, J.P.M.: Satisfiability-based algorithms for boolean optimization. Ann. Math. Artif. Intell. **40** (2004) 353–372
30. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: SAT. (2006) 156–169