ELSEVIER

A logical approach to efficient Max-SAT solving ☆

Javier Larrosa [a,*], Federico Heras [a], Simon de Givry [b]

[a] *Universitat Politecnica de Catalunya, Barcelona, Spain*
[b] *INRA, Toulouse, France*

Abstract

Weighted Max-SAT is the optimization version of SAT and many important problems can be naturally encoded as such. Solving weighted Max-SAT is an important problem from both a theoretical and a practical point of view. In recent years, there has been considerable interest in finding efficient solving techniques. Most of this work focuses on the computation of good quality lower bounds to be used within a branch and bound DPLL-like algorithm. Most often, these lower bounds are described in a procedural way. Because of that, it is difficult to realize the *logic* that is behind.

In this paper we introduce an original framework for Max-SAT that stresses the parallelism with classical SAT. Then, we extend the two basic SAT solving techniques: *search* and *inference*. We show that many algorithmic *tricks* used in state-of-the-art Max-SAT solvers are easily expressible in *logical* terms in a unified manner, using our framework.

We also introduce an original search algorithm that performs a restricted amount of *weighted resolution* at each visited node. We empirically compare our algorithm with a variety of solving alternatives on several benchmarks. Our experiments, which constitute to the best of our knowledge the most comprehensive Max-SAT evaluation ever reported, demonstrate the practical usability of our approach.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Max-SAT; Search; Inference

1. Introduction

Weighted Max-SAT is the optimization version of the SAT problem and many important problems can be naturally expressed as such. They include academic problems such as *Max-Cut* or *Max-Clique*, as well as real problems in domains like *routing* [3], *bioinformatics* [4], *scheduling* [5], *probabilistic reasoning* [6] and *electronic markets* [7]. In recent years, there has been a considerable effort in finding efficient exact algorithms. These works can be divided into theoretical [8–10] and empirical [11–15]. A common drawback of all these algorithms is that in spite of the close relationship between SAT and Max-SAT, they cannot be easily described with *logic* terminology. For instance,

---

the contributions of [11–14] are good quality lower bounds to be incorporated into a *depth-first branch and bound* procedure. These lower bounds are mostly defined in a procedural way and it is very difficult to see the *logic* that is behind the execution of the procedure. This is in contrast with SAT algorithms where the solving process can be easily decomposed into atomic logical steps.

In this paper we introduce an original framework for (weighted) Max-SAT in which the notions of *upper* and *lower bound* are incorporated into the problem definition. Under this framework classical SAT is just a particular case of Max-SAT, and the main SAT solving techniques can be naturally extended. In particular, we extend the basic simplification rules (for example, *idempotency*, *absorption*, *unit clause reduction*, *etc*.) and introduce a new one, *hardening*, that does not make sense in the SAT context. We also extend the two fundamental SAT algorithms: DPLL (based on *search*) and DP (based on *inference*). We also show that the complexity of the extension of DP is exponential on the formula's *induced width* (which is hardly a surprise, since this is also the case of other inference algorithms for graphical models [16,17]). Interestingly, our resolution rule includes, as special cases, many techniques spread over the recent Max-SAT literature. One merit of our framework is that it allows to see all these techniques as inference rules that *transform* the problem into an equivalent simpler one, as it is customary in the SAT context.

The second contribution of this paper is more practical. We introduce an original search algorithm that incorporates three different forms of resolution at each visited node: *neighborhood resolution*, *chain resolution* and *cycle resolution*. Our experimental results on a variety of domains indicate that our algorithm is generally much more efficient than its competitors. This is especially true as the ratio between the number of clauses and the number of variables increases. Note that these are typically the hardest instances for Max-SAT. Our experiments include random weighted and unweighted Max-SAT, Max-One, Max-Cut, Max-Clique, and combinatorial auctions.

The structure of the paper is as follows: In Section 2 we review SAT terminology. In Section 3 we present Max-SAT and introduce our framework. In Section 4 we extend the essential solving techniques from SAT to Max-SAT. Section 5 summarizes in a unified way several specialized forms of resolution that can be used to simplify Max-SAT formula. Section 6 describes our solver. Section 7 reports our experimental work, which corroborate the efficiency of our solver compared to other state-of-the-art solving alternatives. Section 8 discusses related work. Finally, Section 9 concludes and points out directions of future work.

2. Preliminaries on SAT

In the sequel $X = \{x_1, x_2, \ldots, x_n\}$ is a 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 $var(l)$ (namely, $var(x_i) = var(\bar{x}_i) = x_i$). If variable x_i is assigned to *true* literal x_i is satisfied and literal $\bar{x}_i$ is falsified. Similarly, if variable x_i is instantiated to *false*, literal $\bar{x}_i$ is satisfied and literal x_i is falsified. An assignment is *complete* if it gives values to all the variables in X (otherwise it is partial). A *clause* $C = l_1 \vee l_2 \vee \cdots \vee l_k$ is a disjunction of literals such that $\forall_{1 \leqslant i, j \leqslant k, \, i \neq j} \; var(l_i) \neq var(l_j)$. It is customary to think of a clause as a set of literals, which allows to use the usual set operations. If $x \in C$ (resp. $\bar{x} \in C$) we say that x appears in the clause with positive (resp. negative) sign. The size of a clause, noted $|C|$, is the number of literals that it has. $var(C)$ is the set of variables that appear in C (namely, $var(C) = \{var(l) \mid l \in C\}$). An assignment satisfies a clause if and only if it satisfies one or more of its literals. Consequently, the empty clause, noted $\square$, cannot be satisfied. Conversely, a clause which contains the negation of the empty clause, noted $\neg\square$, is always satisfied and can be discarded. Sometimes it is convenient to think of clause C as its equivalent $C \vee \square$. A logical formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a conjunction of different clauses, normally expressed as a set. A satisfying complete assignment is called a *model* of the formula. Given a CNF formula, the SAT problem consists in determining whether there is any model for it or not. The empty formula, noted $\emptyset$, is trivially satisfiable. A formula containing the empty clause is trivially unsatisfiable and we say that it contains an *explicit contradiction*.

2.1. Graph concepts [18]

The structure of a CNF formula $\mathcal{F}$ can be described by its *interaction graph* $G(\mathcal{F})$ containing one vertex associated to each Boolean variable. There is an edge for each pair of vertices that correspond to variables appearing in the same clause. Given a graph G and an ordering of its vertices d, the *parents* of a node x_i is the set of vertices connected to x_i that precede x_i in the ordering. The *width* of x_i along d is the number of parents that it has. The *width of the graph* along d, denoted w_d, is the maximum width among the vertices.
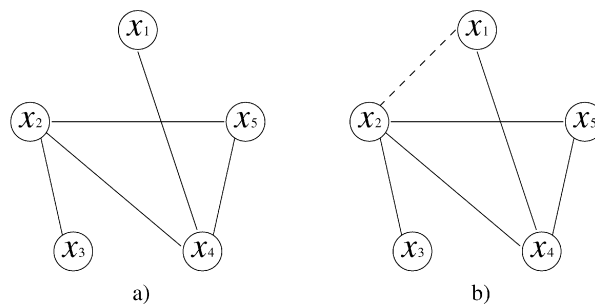
Fig. 1. On the left, a graph G. On the right, the induced graph G_d^* where d is the lexicographic order.

The *induced graph* of $G(\mathcal{F})$ along d, denoted $G_d^*(\mathcal{F})$, is obtained as follows: The vertices of G are processed from last to first along d. When processing vertex x_i, we connect every pair of unconnected parents. The *induced width* of G along d, denoted w_d^*, is the width of the induced graph. The induced width (also known as *tree-width*, *k-tree number* or the *dimension of the graph*) is a measure of how far a graph is from acyclicity and it is a fundamental structural parameter in the characterization of many combinatorial algorithms. Computing the ordering d that provides the minimum induced width is an NP-hard problem [19].

Example 1. Consider the formula $\mathcal{F} = \{\bar{x}_1 \vee x_4, x_1 \vee x_4, x_2 \vee x_3, x_2 \vee x_4, x_2 \vee \bar{x}_5, x_4 \vee x_5\}$. Its interaction graph $G(\mathcal{F})$ is depicted in Fig. 1(a). The induced graph G_d^* along the lexicographical order is depicted in Fig. 1(b). Dotted edge (x_1, x_2) is the only new edge with respect to the original graph. When processing node x_5, no new edges are added, because the parents x_2 and x_4 of x_5 are already connected. When processing node x_4, the edge connecting x_2 and x_1 is added because both variables are parents of x_4 and they were not connected. When processing x_3, x_2 and x_1, no new edges are added. The induced width w_d^* is 2 because nodes x_5 and x_4 have width 2 (namely, they have two parents) in the induced graph.

2.2. SAT algorithms

CNF formulas can be simplified using equivalences or reductions. Well known equivalences are *idempotency* $C \wedge C \equiv C$, *absorption* $C \wedge (C \vee B) \equiv C$ and *unit clause reduction* $l \wedge (\bar{l} \vee C) \equiv l \wedge C$. A well known reduction is the *pure literal rule* which says that if there is a variable that only occurs in either positive or negative form, all clauses mentioning it can be discarded from the formula. Simplifications and reduction have a cascade effect and can be applied until quiescence. The assignment of *true* (resp. *false*) to variable x in $\mathcal{F}$ is noted $\mathcal{F}[x]$ (resp. $\mathcal{F}[\bar{x}]$) and produces a new formula in which all clauses containing x (resp. $\bar{x}$) are eliminated from the formula, and $\bar{x}$ (resp. x) is removed from all clauses where it appears. Note that $\mathcal{F}[l]$ can be seen as the addition of l to the formula and the repeated application of unit clause reduction followed by the pure literal rule.

Algorithms for SAT can be roughly divided into *search* and *inference*. The most popular search algorithm and the starting point of most state-of-the-art SAT solvers was proposed in [20] and is usually called *Davis Putnam Logemann Loveland* (DPLL). Fig. 2 provides a recursive description. First, DPLL simplifies its input (line 1). If the resulting formula is empty, it reports success (line 2). If the resulting formula is a contradiction, it reports failure (line 3). Else it selects a literal l (line 4) and sequentially assigns the formula with l and $\bar{l}$ (line 5).

We say that two clauses $x \vee A, \bar{x} \vee B \in \mathcal{F}$ *clash* if and only if $A \vee B$ is not a tautology and is not absorbed in $\mathcal{F}$. More formally, we define the Clash function:

$$\text{Clash}(x \vee A, \bar{x} \vee B) = \begin{cases} true & \forall_{l \in A} \; \bar{l} \notin B \wedge \forall_{C \in \mathcal{F}} \; C \nsubseteq A \vee B \\ false & \text{otherwise} \end{cases}$$

The *resolution* rule, $\{x \vee A, \bar{x} \vee B\} \equiv \{x \vee A, \bar{x} \vee B, A \vee B\}$, is applied to clashing clauses and is central to inference algorithms. Variable x is called the *clashing variable* and $A \vee B$ is called the *resolvent*. Resolution, which is sound and complete, adds to the formula (i.e., makes explicit) an implicit relation between A and B. Note that unit clause reduction is just a particular case of resolution.

Two years before DPLL, Davis and Putnam proved that a restricted amount of resolution performed along some ordering of the variables is sufficient for deciding satisfiability. The corresponding algorithm is noted DP [18,21].

---

function DPLL($\mathcal{F}$) **return** Boolean
1. $\mathcal{F} := $ Simplify($\mathcal{F}$)
2. **if** $\mathcal{F} = \emptyset$ **then return true**
3. **if** $\mathcal{F} = \{\square\}$ **then return false**
4. $l := $ SelectLiteral($\mathcal{F}$)
5. **return** DPLL($\mathcal{F}[l]$) $\vee$ DPLL($\mathcal{F}[\bar{l}]$)
endfunction

Fig. 2. DPLL is a search algorithm. It returns *true* iff $\mathcal{F}$ is satisfiable.

---

function VarElim($\mathcal{F}, x_i$) **return** CNF formula
1. $\mathcal{B} := \{C \in \mathcal{F} \mid x_i \in var(C)\}$
2. $\mathcal{F} := \mathcal{F} - \mathcal{B}$
3. **while** $\exists\, x_i \vee A \in \mathcal{B}$ **do**
4. $\quad$ $x_i \vee A := $ PopClause($\mathcal{B}$)
5. $\quad$ **while** $\exists_{\bar{x}_i \vee B \in \mathcal{B}}$ s.t. Clash($x_i \vee A, \bar{x}_i \vee B$) **do**
6. $\quad\quad$ $\mathcal{F} := \mathcal{F} \cup \{A \vee B\}$
7. $\quad$ **endwhile**
8. **endwhile**
9. **return** ($\mathcal{F}$)
endfunction
function DP($\mathcal{F}$) **return** Boolean
10. $\mathcal{F} := $ Simplify($\mathcal{F}$)
11. **if** $\mathcal{F} = \emptyset$ **then return true**
12. **if** $\mathcal{F} = \{\square\}$ **then return false**
13. $x_i := $ SelectVar($\mathcal{F}$)
14. **return** DP(VarElim($\mathcal{F}, x_i$))
endfunction

Fig. 3. DP is a pure inference algorithm. It returns *true* iff $\mathcal{F}$ is satisfiable.

Fig. 3 provides a recursive description. It eliminates variables one-by-one until it obtains the empty formula or achieves a contradiction. The heart of DP is Function VarElim. It eliminates variable x_i from formula $\mathcal{F}$ while preserving its solvability. First, it computes the so-called *bucket* of x_i, noted $\mathcal{B}$, which contains the set of clauses mentioning the variable (line 1). All the clauses in the bucket are removed from the formula (line 2). Next, it applies resolution restricted to the clauses in the bucket while pairs of clashing clauses exist. Resolvents are added to the formula (line 6). The correctness of DP is based on the fact that clauses added in line 6 keep the essential information contained in clauses removed in line 2. Observe that the pure literal rule is just a special case of variable elimination in which no pair of clashing clauses exist, so the inner loop never iterates.

The following lemma shows how the complexity of eliminating a variable depends on the number of other variables that it interacts with,

Lemma 2. *(See [18].) Let $\mathcal{F}$ be a CNF formula and x_i one of its variables. Let n_i be the number of variables sharing some clause with x_i in $\mathcal{F}$. The space and time complexity of* VarElim($\mathcal{F}, x_i$) *is* $O(3^{n_i})$ *and* $O(9^{n_i})$, *respectively.*

The following lemma shows how the induced graph $G_d^*(\mathcal{F})$ captures the evolution of the interaction graph $G(\mathcal{F})$ as variables are eliminated.

Lemma 3. *(See [18].) Let d denote the reverse order in which DP($\mathcal{F}$) eliminates variables. The width of x_i along d in the induced graph $G(\mathcal{F})_d^*$ bounds from above the number of variables sharing some clause with x_i at the time of its elimination.*

Thus, the induced width captures the most expensive variable elimination. The following theorem, which follows from the two previous lemmas, characterizes the complexity of DP in terms of the induced width.

Theorem 4. *(See [18].) Let d denote the reverse order in which $DP(\mathcal{F})$ eliminates variables. Let w_d^* denote the induced width of $G(\mathcal{F})$ along d. The space and time complexity of $DP(\mathcal{F})$ is $O(n \times 3^{w_d^*})$ and $O(n \times 9^{w_d^*})$, respectively.*

A consequence of the previous theorem is that the order in which DP eliminates variables may be crucial for the algorithm's complexity. As an example, consider a formula, whose interaction graph is a tree of depth 1. If variables are eliminated in a top-down order, the cost may be exponential in n. If variables are eliminated in a bottom-up order, the cost is linear. In general, finding optimal elimination orderings is an NP-hard problem and approximate algorithms must be used. In practical applications, DP is generally too space consuming and cannot be used [18]. Nevertheless, resolution still plays an important practical role in combination with search: the addition of restricted forms of resolution at each search node anticipates the detection of dead-ends and improves its performance [18, 22–24]. As we will show, the use of resolution is even more relevant in the Max-SAT context.

3. (Weighted) Max-SAT

When a Boolean formula does not have any model, one may be interested in finding a complete assignment with minimum number of falsified clauses. This problem is known as (*unweighted*) *Max-SAT*. Note that no repetition of clauses is allowed and all clauses are equally important. The complexity of Max-SAT is $P^{NP[\log n]}$, meaning that it can be solved with a logarithmic number of calls to a SAT oracle [25].

Weighted Max-SAT is an extension of Max-SAT. A *weighted clause* is a pair (C, w) such that C is a classical clause and w is a natural number indicating the cost of its falsification. A weighted formula in conjunctive normal form is a *set* of weighted clauses. The *cost* of an assignment is the sum of weights of all the clauses that it falsifies. Given a weighted formula, *weighted* Max-SAT is the problem of finding a complete assignment with minimal cost. We can assume all clauses in the formula being different, since $(C, u), (C, w)$ can be replaced by $(C, u + w)$. Note that clauses with cost 0 do not have any effect and can be discarded. Weighted Max-SAT is more expressive than unweighted Max-SAT and its complexity, P^{NP}, is higher [25] (it may require a linear number of calls to a SAT oracle). Since most Max-SAT applications require the expressiveness of weights, in this paper we will focus on weighted Max-SAT. In the following, when we say Max-SAT we will be referring to *weighted* Max-SAT.

Example 5. 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 is a well-known NP-hard problem. It 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. Value *true* (respectively, *false*) indicates that vertex x_i belongs to U (respectively, to $V - U$). There is a binary weighted clause $(x_i \vee x_j, u)$ for each edge $(v_i, v_j) \in E$, where u is a number larger than or equal to $|V|$. It specifies that at least one of these vertices must 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. Note that different weights in unary and binary clauses are required to express the relative importance of each type of clauses.

Consider the minimum vertex covering of the graph in Fig. 1(a). The Max-SAT encoding is $\mathcal{F} = \{(\bar{x}_1, 1), (\bar{x}_2, 1), (\bar{x}_3, 1), (\bar{x}_4, 1), (\bar{x}_5, 1), (x_1 \vee x_4, 5), (x_2 \vee x_3, 5), (x_2 \vee x_4, 5), (x_2 \vee x_5, 5), (x_4 \vee x_5, 5)\}$. The optimal assignment is $\{x_2 = x_4 = true, x_1 = x_3 = x_5 = false\}$ with cost 2 that is equal to the size of the minimum vertex covering.

Next, we propose an alternative, although equivalent, definition for weighted Max-SAT that will be more convenient for our purposes. Given a weighted CNF formula, we assume the existence of a known upper bound $\top$ on the cost of an optimal solution ($\top$ is a strictly positive natural number). This is done without loss of generality because, if a tight upper bound is not known, $\top$ can be set to any number higher than the sum of weights of all the clauses. A *model* for the formula is a complete assignment with cost less than $\top$. An *optimal model* is a model of minimal cost. Then, Max-SAT can be reformulated as the problem of finding an optimal model, if there is any. Observe that any weight $w \geqslant \top$ indicates that the associated clause *must be necessarily satisfied*. Thus, we can replace w by $\top$ without changing the problem. Thus, without loss of generality we assume all costs in the interval $[0..\top]$ and define the *sum of costs* as,

$$a \oplus b = \min\{a + b, \top\}$$

in order to keep the result within the interval $[0..\top]$. A clause with cost $\top$ is called *mandatory* (or *hard*). A clause with cost less than $\top$ is called *non-mandatory* (or *soft*).

Definition 6. A Max-SAT instance is a pair $(\mathcal{F}, \top)$ where $\top$ is a natural number and $\mathcal{F}$ is a set of weighted clauses with weights in the interval $[0..\top]$. The task of interest is to find an optimal model, if there is any.

The following example shows that $\top$ can be used to express that we are only interested in assignments of a certain quality.

Example 7. Consider again the minimum vertex covering problem of the graph in Fig. 1(a). With the new notation, the associated formula is

$$\mathcal{F} = \big\{ (\bar{x}_1, 1), (\bar{x}_2, 1), (\bar{x}_3, 1), (\bar{x}_4, 1), (\bar{x}_5, 1), (x_1 \vee x_4, \top), (x_2 \vee x_3, \top), (x_2 \vee x_4, \top), (x_2 \vee x_5, \top), (x_4 \vee x_5, \top) \big\}$$

which shows more clearly which clauses are truly weighted and which ones are mandatory. In the lack of additional information, $\top$ should be set to the sum of weights ($\top = 5$), meaning that any assignment that satisfies the mandatory clauses should be taken into consideration. Suppose now that somehow (for example, with a local search algorithm) we find a covering of size 3. We can set $\top$ to 3 because any assignment with cost 3 or higher does not interest us anymore. The resulting Max-SAT problem is tighter (and easier, because more partial assignments can be identified as being unfeasible).

The interest of adding $\top$ to the problem formulation is twofold. On the one hand, it makes explicit the mandatory nature of mandatory clauses. Besides, as we will see later, it allows to *discover* mandatory clauses that were *disguised* as weighted clauses. On the other hand, it allows to see SAT as a particular case of Max-SAT.

Remark 8. A Max-SAT instance with $\top = 1$ is essentially a SAT instance because there is no weight below $\top$. Consequently, every clause in the formula is mandatory.

A weighted CNF formula may contain $(\square, w)$ among its clauses. Since $\square$ cannot be satisfied, w is a necessary cost of any model. Therefore, w is an explicit *lower bound* of the cost of an optimal model. When the lower bound and the upper bound have the same value (namely, $(\square, \top) \in \mathcal{F}$) the formula is trivially unsatisfiable and we call this situation an *explicit contradiction*.

4. Extending SAT solving techniques to Max-SAT

4.1. Extending simplification rules and clause negation

We say that two Max-SAT formulas are equivalent, $\mathcal{F} \equiv \mathcal{F}'$, if the cost of their optimal assignment is the same or if neither of them has a model. The following equivalence rules can be used to simplify CNF weighted formulas,

- *Aggregation*: $\{(A, w), (A, u)\} \equiv \{(A, w \oplus u)\}$.
- *Absorption*: $\{(A, \top), (A \vee B, w)\} \equiv \{(A, \top)\}$.
- *Unit clause reduction*: $\{(l, \top), (\bar{l} \vee A, w)\} \equiv \{(l, \top), (A, w)\}$.
- *Hardening*: If $\bigoplus_{i=1}^{k} u_i = \top$ and $\forall_{1 \leqslant i < k} C_i \subset C_k$ then

$$\big\{(C_i, u_i)\big\}_{i=1}^{k-1} \cup \big\{(C_k, u_k)\big\} \equiv \big\{(C_i, u_i)\big\}_{i=1}^{k-1} \cup \big\{(C_k, \top)\big\}.$$

Aggregation generalizes to Max-SAT the idempotency of the conjunction in classical SAT. The *Absorption* rule indicates that in the Max-SAT context the absorbing clause must be mandatory. Similarly, *unit clause reduction* requires the unit clause being mandatory. The correctness of these equivalences is direct and we omit the proof. The *Hardening* rule allows to identify weighted clauses that are indeed mandatory. It holds because the violation of C_k implies the violation of all C_i with $i < k$. Therefore, any assignment that violates C_k will have cost $\bigoplus_{i=1}^{k} u_i = \top$.

It is easy to see that the definitions of the *pure literal rule* and the assignment of a formula $\mathcal{F}[l]$ (see Section 2.2), can be directly applied to Max-SAT. As in SAT, $\mathcal{F}[l]$ can be seen as the addition of $(l, \top)$ to the formula which allows a sequence of unit clause reductions followed by the application of the pure literal rule.

Example 9. Consider the following formula $\{(x, \top), (\bar{x}, 3), (y, 8), (\bar{x} \vee \bar{y}, 3)\}$ with $\top = 10$. We can apply unit clause reduction to the first and second clauses, which produces $\{(x, \top), (\square, 3), (y, 8), (\bar{x} \vee \bar{y}, 3)\}$. We can apply it again to the first and fourth clauses producing $\{(x, \top), (\square, 3), (y, 8), (\bar{y}, 3)\}$. The pure literal rule allows to remove the first clause producing $\{(\square, 3), (y, 8), (\bar{y}, 3)\}$. We can harden the second clause because $3 \oplus 8 = \top$. Thus, we obtain $\{(\square, 3), (y, \top), (\bar{y}, 3)\}$. Unit clause reduction produces $\{(\square, 3), (y, \top), (\square, 3)\}$. Aggregation yields $\{(\square, 6), (y, \top)\}$ and the pure literal rule produces the formula $\{(\square, 6)\}$ which trivially has an optimal model of cost 6.

Proposition 10. *The algorithm that applies the previous simplifications until quiescence terminates in polynomial time.*

Observe that if an explicit contradiction is achieved (i.e., $(\square, \top) \in \mathcal{F}$), all clauses are subsequently absorbed and the formula immediately collapses to $(\square, \top)$.

The *negation of a weighted clause* (C, w), noted $(\bar{C}, w)$, means that the satisfaction of C has cost w, while its falsification is cost-free. Note that $\bar{C}$ is not in clausal form when $|C| > 1$. In classical SAT the *De Morgan* rule can be used to recover the CNF syntax, but the following example shows that it cannot be applied to weighted clauses.

Example 11. Consider the weighted clause $(x \vee y, 1)$ with $\top > 1$. The truth table of its negation $(\overline{x \vee y}, 1)$ and the truth table of $\{(\bar{x}, 1), (\bar{y}, 1)\}$ are given below (ignore the last column for the moment). Note that they are not equivalent.

$x\ y$	$(\overline{x \vee y}, 1)$	$\{(\bar{x}, 1), (\bar{y}, 1)\}$	$\{(\bar{x} \vee y, 1), (x \vee \bar{y}, 1), (\bar{x} \vee \bar{y}, 1)\}$
f f	0	$0 \oplus 0 = 0$	$0 \oplus 0 \oplus 0 = 0$
f t	1	$1 \oplus 0 = 1$	$0 \oplus 1 \oplus 0 = 1$
t f	1	$0 \oplus 1 = 1$	$1 \oplus 0 \oplus 0 = 1$
t t	1	$1 \oplus 1 = 2$	$0 \oplus 0 \oplus 1 = 1$

The following recursive transformation rule allows to compute the clausal form for totally or partially negated clauses. Let A and B be arbitrary disjunctions of clauses,

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

The last column in the truth table of the previous example shows the proper CNF encoding of clause $(\overline{x \vee y}, 1)$. The main drawback of this rule is that it generates an exponential number of new clauses with respect to the arity of the negated clause. We will show in Section 4.3 that it is possible to transform it into a linear number of clauses.

Theorem 12. $CNF(A \vee \overline{l \vee B}, u)$ *returns an equivalent CNF expression.*

Proof. It is clear that $CNF(A \vee \overline{l \vee B}, u)$ generates a CNF expression because the negation is applied to a smaller sub-expression at each recursive call. Eventually, it will be applied to literals, so the expression will be a clause. We prove that $CNF(A \vee \overline{l \vee B}, u)$ returns an equivalent expression by induction over $|B|$. The $|B| = 0$ is trivial since the left-hand and the right-hand sides are the same. Regarding the $|B| > 0$ case, there are three ways to falsify $A \vee \overline{l \vee B}$. Each one of the three elements in the right-hand side corresponds to one of them. The last two are assumed correct by the induction hypothesis. $\square$

Remark 13. The weighted expression $(A \vee C \vee (\overline{C \vee B}), u)$, where A, B and C are disjunctions of literals, is equivalent to $(A \vee C \vee \bar{B}, u)$, because they are falsified under the same circumstances.

```
function Max-DPLL(F, ⊤) return ℕ
1.  F := Simplify(F, ⊤)
2.  if F = ∅  then return 0
3.  if F = {(□, w)}  then return w
4.  l := SelectLiteral(F)
5.  ⊤ := Max-DPLL(F[l], ⊤)
6.  ⊤ := Max-DPLL(F[l̄], ⊤)
7.  return ⊤
endfunction
```

Fig. 4. If $(\mathcal{F}, \top)$ has models, Max-DPLL returns the optimal cost. Else it returns $\top$.

4.2. Extending DPLL

In Fig. 4 we present Max-DPLL, the extension of DPLL to Max-SAT. Max-DPLL$(\mathcal{F}, \top)$ returns the cost of the optimal model if there is any, else it returns $\top$. First, the input formula is simplified with the rules from the previous subsection (line 1). If the resulting formula is empty, there is a 0 cost model (line 2). If the resulting formula only contains the empty clause, the algorithm returns its cost (line 3). Else, it selects a literal l (line 4) and makes two recursive calls (lines 5 and 6). In each call, the formula is instantiated with l and $\bar{l}$. Observe that the first recursive call is made with the $\top$ inherited from its parent, but the second call uses the output of the first call. This implements the typical upper bound updating of depth-first branch and bound. Finally, the best value of the two recursive calls is returned (line 7). Observe that, as search goes on, the value of $\top$ may decrease. Consequently, clauses that originally were soft may become hard which, in turn, may strengthen the potential of the simplification rules. The parallelism with DPLL (Fig. 2) is obvious. The following statement shows that Max-DPLL is a true extension of classical DPLL.

Remark 14. The execution of Max-DPLL with a SAT instance (i.e., $(\mathcal{F}, \top)$ with $\top = 1$) behaves like classical DPLL.

It is easy to see that the time complexity of Max-DPLL is exponential on the number of variables n and the space complexity is polynomial on $|\mathcal{F}|$. Therefore, DPLL and Max-DPLL have the same complexity.

4.3. Extending the resolution rule

Consider the *subtraction* of costs ($\ominus$) defined as in [26]. Let $u, w \in [0, \ldots, \top]$ be two weights such that $u \geqslant w$,

$$u \ominus w = \begin{cases} u - w, & u \neq \top \\ \top, & u = \top \end{cases}$$

Essentially, $\ominus$ behaves like the usual subtraction except that $\top$ is an absorbing element. The resolution rule can be extended from SAT to Max-SAT as,

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

where $m = \min\{u, w\}$. In this rule, that we call Max-RES, $(A \vee B, m)$ is called the *resolvent*; $(x \vee A, u \ominus m)$ and $(\bar{x} \vee B, w \ominus m)$ are called the *posterior clashing clauses*. $(x \vee A \vee \bar{B}, m)$ and $(\bar{x} \vee \bar{A} \vee B, m)$ are called the *compensation clauses*. The effect of Max-RES, 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-RES. While classical resolution 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 subtracted in order to *compensate* the new inferred information. Consequently, Max-RES is better understood as a *movement* of knowledge.

Example 15. If we apply Max-RES to the following clauses $\{(x \vee y, 3), (\bar{x} \vee y \vee z, 4)\}$ (with $\top > 4$) we obtain $\{(y \vee y \vee z, 3), (x \vee y, 3 \ominus 3), (\bar{x} \vee y \vee z, 4 \ominus 3), (x \vee y \vee \overline{(y \vee z)}, 3), (\bar{x} \vee \bar{y} \vee y \vee z, 3)\}$. The first and fourth clauses can be simplified (see Remark 13). The second clause can be omitted because its weight is zero. The fifth clause can be omitted because it is a tautology. Therefore, we obtain the equivalent formula $\{(y \vee z, 3), (\bar{x} \vee y \vee z, 1), (x \vee y \vee \bar{z}, 3)\}$.

The previous example shows that, under certain conditions, some of the right-hand side clauses can be removed. Clause $(x \vee A, u \ominus m)$ (symmetrically for $(\bar{x} \vee B, w \ominus m)$) can be omitted if and only if either,

- $B \subseteq A \wedge m = \top$, or
- $u = m < \top$.

The first case holds because the clause is absorbed by the resolvent $(A, \top)$. The second case holds because $u \ominus m = 0$.
Regarding clause $(x \vee A \vee \bar{B}, m)$ (symmetrically for $(\bar{x} \vee \bar{A} \vee B, m)$), it can be omitted if and only if either,

- $B \subseteq A$, or
- $u = \top$.

The first case holds because the clause is a tautology. The second case holds because the clause is absorbed by the posterior clashing clause $(x \vee A, \top \ominus m = \top)$.

Remark 16. The application of Max-RES to mandatory clauses is equivalent to classical resolution.

Proof. Clashing clauses being mandatory means that $u = w = \top$. Clearly, $m = \min\{u, w\} = \top$, $u \ominus m = \top$ and $w \ominus m = \top$. Consequently, all right-hand clauses are mandatory. Therefore, the prior and posterior clashing clauses are equal. Furthermore, the compensation clauses are absorbed by the clashing clauses (as we previously noted). Thus, Max-RES has the effect of adding $(A \vee B, \top)$ to the formula, which is equivalent to classical resolution. $\quad\square$

Theorem 17. *Max-RES is sound.*

Proof. The following table contains in the first column all the truth assignments, in the second column the cost of the assignment according to the clauses on the left-hand of the Max-RES rule, and in the third column the cost of the assignment according to the clauses on the right-hand of the Max-RES rule. As it can be observed, the costs are the same, so the resulting problem is equivalent.

$x\ A\ B$	Left	Right
f f f	u	$m \oplus (u \ominus m)$
f f t	u	$m \oplus (u \ominus m)$
f t f	0	0
f t t	0	0
t f f	w	$m \oplus (w \ominus m)$
t f t	0	0
t t f	w	$m \oplus (w \ominus m)$
t t t	0	0

Observe that compensation clauses $(x \vee A \vee \bar{B}, m)$ and $(\bar{x} \vee \bar{A} \vee B, m)$ are not in clausal form when $|A| > 1$ and $|B| > 1$. In the following, we assume that they are transformed to clausal forms as needed. In Section 4.1, we introduced a recursive rule that computes a clausal form for totally or partially negated clauses. We noted that it produces an exponentially large number of new clauses. Interestingly, using insights from the Max-RES rule, we can redefine it in such a way that only a linear number of clauses is generated,

$$CNF_{\texttt{linear}}(A \vee \overline{l \vee B}, u) = \begin{cases} A \vee \bar{l}, & |B| = 0 \\ \{(A \vee \bar{l} \vee B, u)\} \cup CNF_{\texttt{linear}}(A \vee \bar{B}, u), & |B| > 0 \end{cases}$$

The new rule is correct because the two recursive calls of CNF (Section 4.1), $CNF(A \vee l \vee \bar{B}, u)$ and $CNF(A \vee \bar{l} \vee \bar{B}, u)$, can be resolved on literal l and we obtain the equivalent call $CNF(A \vee \bar{B}, u)$. For example, the application

function `Max-VarElim`$(\mathcal{F}, \top, x_i)$ **return** weighted CNF formula
1. $\mathcal{B} := \{(C, u) \in \mathcal{F} \mid x_i \in var(C)\}$
2. $\mathcal{F} := \mathcal{F} - \mathcal{B}$
3.   **while** $\exists (x_i \vee A, u) \in \mathcal{B}$ **do**
4. $(x_i \vee A, u) := $ `PopMinSizeClause`$(\mathcal{B})$
5.   **while** $u > 0 \wedge \exists_{(\bar{x}_i \vee B, w) \in \mathcal{B}}$ s.t. `Clash`$(x_i \vee A, \bar{x}_i \vee B)$ **do**
6. $m := \min\{u, w\}$
7. $u := u \ominus m$
8. $\mathcal{B} := \mathcal{B} - \{(\bar{x}_i \vee B, w)\} \cup \{(\bar{x}_i \vee B, w \ominus m)\}$
9. $\mathcal{B} := \mathcal{B} \cup \{(x_i \vee A \vee \bar{B}, m), (\bar{x}_i \vee \bar{A} \vee B, m)\}$
10. $\mathcal{F} := \mathcal{F} \cup \{(A \vee B, m)\}$
11.   **endwhile**
12. **endwhile**
13. **return** $(\mathcal{F})$
endfunction
function `Max-DP`$(\mathcal{F}, \top)$ **return** $\mathbb{N}$
14. $\mathcal{F} := $ `Simplify`$(\mathcal{F}, \top)$
15. **if** $\mathcal{F} = \emptyset$ **then return** 0
16. **if** $\mathcal{F} = \{(\square, u)\}$ **then return** u
17. $x_i := $ `SelectVar`$(\mathcal{F})$
18. **return** `Max-DP`(`VarElim`$(\mathcal{F}, \top, x_i), \top)$
endfunction

Fig. 5. If $(\mathcal{F}, \top)$ has models, Max-DP returns their optimal cost. Else it returns $\top$.

of $CNF_{\texttt{linear}}$ to $(\overline{x \vee y}, 1)$ (Example 11) produces the equivalent $\{(\bar{x} \vee y, 1), (\bar{y}, 1)\}$. Observe that the output of $CNF_{\texttt{linear}}$ depends on how the literals are ordered in the clause. $\square$

4.4. Extending DP

The following example shows that, unlike classical resolution, the unrestricted application of Max-RES does not guarantee termination.

Example 18. Consider the following formula $\{(x \vee y, 1), (\bar{x} \vee z, 1)\}$ with $\top = 3$. If we apply Max-RES, we obtain $\{(y \vee z, 1), (x \vee y \vee \bar{z}, 1), (\bar{x} \vee \bar{y} \vee z, 1)\}$. If we apply Max-RES to the first and second clauses we obtain $\{(x \vee y, 1), (\bar{x} \vee y \vee z, 1), (\bar{x} \vee \bar{y} \vee z, 1)\}$. If we apply now Max-RES to the second and third clauses we obtain $\{(x \vee y, 1), (\bar{x} \vee z, 1)\}$, which is the initial formula.

Nevertheless, Bonet et al. [27] have recently proved that when all clauses are non-mandatory, the directional application of Max-RES solves the Max-SAT problem. If their proof is combined with the proof of correctness of DP [21] (namely, all clauses being mandatory), we have that the extension of DP to Max-SAT produces a correct algorithm. Max-DP (depicted in Fig. 5) is the extension of DP to Max-SAT. Both algorithms are essentially equivalent the main difference being that Max-DP performs Max-RES instead of classical resolution. Observe the parallelism between Function `VarElim` (Fig. 3) and Function `Max-VarElim` (Fig. 5). Both are in charge of the elimination of variable x_i from the formula. As in the SAT case, `Max-VarElim` computes the bucket $\mathcal{B}$ (line 1) and removes its clauses from the formula (line 2). Then, it selects a clause $(x \vee A, u)$ and resolves it with all its clashing clauses. The function `Clash` is similar to its SAT definition (see Section 2.2), that is $A \vee B$ is not a tautology and is not absorbed in $\mathcal{F}$, except that it is now based on the Max-SAT definition of absorption. In `VarElim`, clause $x \vee A$ is resolved until no clashing clauses exist. In `Max-VarElim`, clause $(x \vee A, u)$ is resolved until its weight u decreases to 0 or no clashing clauses exist. A difference worth noting with respect to the SAT case is that `Max-VarElim` selects in line 4 a *minimal size* clause. This is not required for the correctness of the algorithm but only to achieve the complexity stated in Theorem 23.

The following lemma shows that `Max-VarElim` transforms the input formula preserving its optimality.

Lemma 19. *Consider a call to the* `Max-VarElim` *function. Let* $(\mathcal{F}, \top)$ *denote the input formula and let* $(\mathcal{F}', \top)$ *denote the output formula. It is true that* $(\mathcal{F}, \top)$ *has models if and only if* $(\mathcal{F}', \top)$ *has models. Besides, if* $(\mathcal{F}, \top)$ *has models, the cost of the optimal one is the same as the cost of the optimal model of* $(\mathcal{F}', \top)$.

Proof. See Appendix A. □

Theorem 20. *Algorithm Max-DP is correct.*

Proof. Max-DP is a sequence of variable eliminations until a variable-free formula is obtained. Lemma 19 shows that each transformation preserves the cost of the optimal model. Therefore, the cost of the final variable-free formula $(\Box, u)$ is the cost of the optimal model of the original formula. □

The following lemma, shows that the complexity of eliminating a variable is the same in classical SAT as in Max-SAT.

Lemma 21. *Let* $(\mathcal{F}, \top)$ *be a Max-SAT instance and* x_i *one of its variables. Let* n_i *denote the number of variables sharing some clause with* x_i *in* $\mathcal{F}$. *The space and time complexity of* `Max-VarElim`$(\mathcal{F}, \top, x_i)$ *is* $O(3^{n_i})$ *and* $O(9^{n_i})$, *respectively.*

Proof. See Appendix A. □

The next lemma shows that the induced graph plays the same role in DP as in Max-DP.

Lemma 22. *Let* d *denote the reverse order in which Max-DP*$(\mathcal{F}, \top)$ *eliminates variables. The width of* x_i *along* d *in the induced graph* $G(\mathcal{F})_d^*$ *bounds above the number of variables sharing some clause with* x_i *at the time of its elimination.*

Proof. Same as in the SAT case (Lemma 3). □

The following theorem, which trivially follows from the previous two lemmas, bounds the complexity of Max-DP.

Theorem 23. *Let* $(\mathcal{F}, \top)$ *be an arbitrary Max-SAT instance. Let* d *denote the reverse order in which Max-DP*$(\mathcal{F}, \top)$ *eliminates variables. The space and time complexity of DP*$(\mathcal{F})$ *is* $O(n \times 3^{w_d^*})$ *and* $O(n \times 9^{w_d^*})$, *respectively, where* w_d^* *is the induced width of the interaction graph* $G(\mathcal{F})$ *along* d.

Observe that the complexities of DP and Max-DP are the same, even though Max-SAT has a complexity higher than SAT. The following remark shows that Max-DP is a true extension of DP.

Remark 24. The execution of Max-DP with a SAT instance (i.e., $(\mathcal{F}, \top)$ with $\top = 1$) behaves like classical DP.

5. Efficient inference

The complexity results of the previous section show that solving Max-SAT with pure resolution methods is in general too space consuming and can only be used in practice with formulas with a small induced width (around 30 with current computers). A natural alternative is to use only restricted forms of resolution that simplify the formula and use search afterwards. In this Section we summarize some simplification rules that have been proposed in the recent Max-SAT literature and show that they can be naturally explained with our framework. We also introduce two original ones that will be used in the solver that we will introduce in Section 6.

We classify these simplification rules in three categories: single applications of resolution, multiple applications of resolution (namely, hyper-resolution), and variable elimination. The following rules are presented in their general form. In Section 6, we will use a subset of these rules in a restricted form.

5.1. Single resolution

Proposition 25. *Unit clause reduction* (*also called upper bound rule in* [13])*,*

$$\big\{(l, \top), (\bar{l} \vee A, w)\big\} \equiv \big\{(l, \top), (A, w)\big\}$$

is a particular case of Max-RES.

Proof. If $w = \top$, we have the classical SAT case, which is trivial. If $w < \top$, we have that the application of Max-RES to $\{(l \vee \square, \top), (\bar{l} \vee A, w)\}$ produces $\{(A, w), (l, \top \ominus w), (\bar{l} \vee A, w \ominus w), (l \vee \square \vee \bar{A}, w), (\bar{l} \vee \neg\square \vee A, w)\}$.

The third clause can be removed because $w \ominus w = 0$. The fourth clause can be removed because it is absorbed by the second. The fifth clause can be removed because it is a tautology. $\square$

Proposition 26. Neighborhood resolution [1] (*also called* replacement of almost common clauses *in* [8])*,*

$$\big\{(l \vee A, u), (\bar{l} \vee A, w)\big\} \equiv \big\{(A, w), (l \vee A, u \ominus w)\big\}$$

where, without loss of generality, $w \leqslant u$, is a particular case of Max-RES.

Proof. Resolving the two left-hand clauses, we obtain $\{(A, w), (l \vee A, u \ominus w), (\bar{l} \vee A, w \ominus w), (l \vee A \vee \bar{A}, w), (\bar{l} \vee \bar{A} \vee A, w)\}$. The third clause can be omitted because either its weight is 0 (when $w < \top$), or it is absorbed by the resolvent (when $w = \top$). The fourth and fifth clauses can be omitted because they are tautologies. $\square$

The simplification potential of neighborhood resolution is shown in the following example.

Example 27. Consider the formula $\{(y \vee z, 1), (\bar{y} \vee z, 1), (\bar{z}, 1)\}$. The application of neighborhood resolution yields $\{(z, 1), (\bar{z}, 1)\}$ which allows a new application of neighborhood resolution producing the trivial formula $\{(\square, 1)\}$.

The term *neighborhood resolution* was coined by [28] in the SAT context. The Max-SAT extension was first proposed in [8]. The practical efficiency of the $|A| = 0, 1, 2$ cases was assessed in [29,30], [14] and [1], respectively.

5.2. Variable elimination

Proposition 28. *The* pure literal rule (*first proposed in the Max-SAT context in* [8]) *is a special case of* `Max-VarElim`*.*

Proof. Consider a formula $\mathcal{F}$ such that there is a literal l, whose negation does not appear in the formula. Let $x = var(l)$. Function `Max-VarElim`$(\mathcal{F}, \top, x)$ has the same effect as the pure literal rule, because there is no pair of clauses clashing on x. Thus, no resolution will be performed and all clauses containing l will be removed from the formula. $\square$

Proposition 29. *The* elimination *rule* [8] (*also called* resolution *in* [9,10]) *which says that if* $\mathcal{F} = \{(l \vee A, u), (\bar{l} \vee B, w)\} \cup \mathcal{F}'$ *and var(l) does not occur in $\mathcal{F}'$ then*

$$\mathcal{F} \equiv \mathcal{F}' \cup \big\{\big(A \vee B, \min\{u, w\}\big)\big\}$$

is a special case of `Max-VarElim`*.*

Proof. Let x be the clashing variable (namely, $x = var(l)$). We need to prove that Function `Max-VarElim` with x as the elimination variable replaces $\{(l \vee A, u), (\bar{l} \vee B, w)\}$ by $\{(A \vee B, \min\{u, w\})\}$. There are two possibilities: If $\{(l \vee A, u), (\bar{l} \vee B, w)\}$ clash, they will be resolved and $(A \vee B, \min\{u, w\})$ will be added to the formula. All the clauses in the bucket after the resolution step do not clash on x, so `Max-VarElim` will discard them. If $\{(l \vee A, u), (\bar{l} \vee B, w)\}$ do not clash, `Max-VarElim` will directly discard them. In that case, $A \vee B$ either is a tautology or is absorbed, so it has no effect on the right-hand side of the elimination rule. $\square$
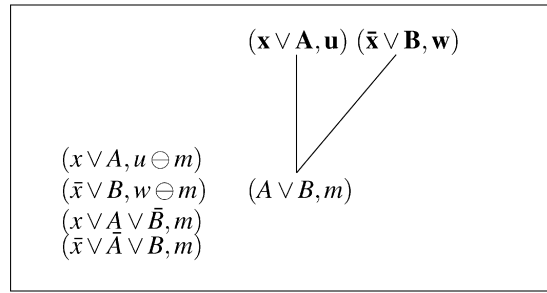
$$(\mathbf{x} \vee \mathbf{A}, \mathbf{u}) \ (\bar{\mathbf{x}} \vee \mathbf{B}, \mathbf{w})$$

$(x \vee A, u \ominus m)$
$(\bar{x} \vee B, w \ominus m)$ $\qquad (A \vee B, m)$
$(x \vee A \vee \bar{B}, m)$
$(\bar{x} \vee \bar{A} \vee B, m)$

Fig. 6. Graphical representation of Max-RES.

Proposition 30. *Let $\tilde{x}$ denote either x or $\bar{x}$. The* small subformula rule [9]*, which says that, if $\mathcal{F} = \{(\tilde{x} \vee \tilde{y} \vee A, u),$ $(\tilde{x} \vee \tilde{y} \vee B, w), (\tilde{x} \vee \tilde{y} \vee C, v)\} \cup \mathcal{F}'$ and x, y do not occur in $\mathcal{F}'$ then*

$$\mathcal{F} \equiv \mathcal{F}'$$

is a special case of `Max-VarElim`*.*

Proof. We only need to prove that if we eliminate x and y from $\{(\tilde{x} \vee \tilde{y} \vee A, u), (\tilde{x} \vee \tilde{y} \vee B, w), (\tilde{x} \vee \tilde{y} \vee C, v)\}$ with function `Max-VarElim`, we obtain the empty formula $\varnothing$.

If all the occurrences of x or y have the same sign, the rule holds because the pure literal rule can be applied. If there are occurrences of different sign, there are only two cases to consider (all other cases are symmetric):

- If we have $\{(x \vee y \vee A, u), (x \vee y \vee B, v), (\bar{x} \vee \bar{y} \vee C, w)\}$, there are no clauses clashing on x (neither on y), so `Max-VarElim` will just discard the clauses.
- If we have $\{(x \vee y \vee A, u), (\bar{x} \vee y \vee B, v), (x \vee \bar{y} \vee C, w)\}$, the first and second clauses clash, so Max-RES produces,

$$\big\{(y \vee A \vee B, m), (x \vee y \vee A, u \ominus m), (\bar{x} \vee y \vee B, v \ominus m), (x \vee y \vee A \vee \overline{y \vee B}, m),$$
$$(\bar{x} \vee \overline{y \vee A} \vee y \vee B, m), (x \vee \bar{y} \vee C, w), \big\}$$

where $m = \min\{u, v\}$; which is equivalent to,

$$\big\{(y \vee A \vee B, m), (x \vee y \vee A, u \ominus m), (\bar{x} \vee y \vee B, v \ominus m), (x \vee y \vee A \vee \bar{B}, m),$$
$$(\bar{x} \vee \bar{A} \vee y \vee B, m), (x \vee \bar{y} \vee C, w)\big\}.$$

There are no further clauses clashing on x. Note that the second and third clauses do not clash: either $u < \top \vee v < \top$ and one of these two clauses has a zero weight and can be discarded, or $u = v = \top$ and because the resolvent is already in the formula (first clause), the two clauses do not clash by definition of absorption. So `Max-VarElim` will just discard all the clauses that mention it, producing the equivalent $\{(y \vee A \vee B, m)\}$. The pure literal rule will eliminate the clause, producing the empty formula. $\quad\square$

5.3. Hyper-resolution

Hyper-resolution is a well known SAT concept that refers to the compression of several resolution steps into one single step. In the following, we introduce four hyper-resolution inference rules. The first two (*star rule* and *dominating unit-clause*) are formal descriptions of already published rules. The other two rules (*cycle* and *chain resolution*) are original. We prove the correctness of these rules by developing the resolution tree that allows to transform the left-hand side of the rule into the right-hand side. Fig. 6 shows the graphical representation of Max-RES. On top there are the two prior clashing clauses. We write them in bold face to emphasize that they are removed from the formula. The resolvent is linked to the prior clashing clauses. At the left of the resolvent, we write the posterior clashing clauses and the compensation clashing clauses, which must be added to preserve equivalence.
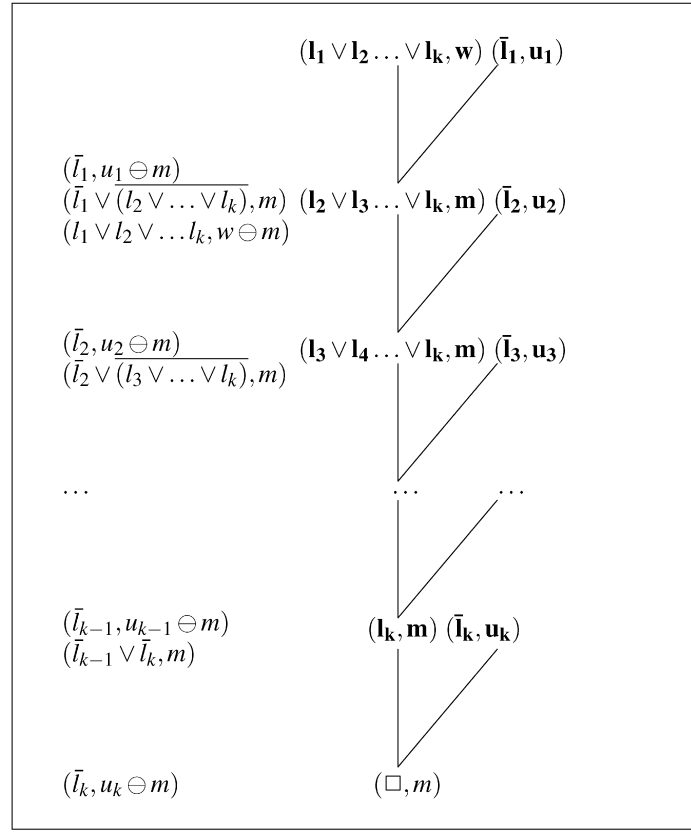
Fig. 7. Resolution tree for the *star rule*.

5.3.1. Star rule

The *star rule* [9,14] identifies a clause of length k such that each of its literals appears negated in a unit clause. Then, at least one of the clauses will be falsified. Formally,

$$\left\{ \begin{array}{l} (l_1 \vee l_2 \vee \cdots \vee l_k, w), \\ (\bar{l}_i, u_i)_{1 \leqslant i \leqslant k} \end{array} \right\} \equiv \left\{ \begin{array}{l} (l_1 \vee l_2 \vee \cdots \vee l_k, w \ominus m), \\ (\bar{l}_i \vee \overline{(l_{i+1} \vee l_{i+2} \vee \cdots \vee l_k)}, m)_{1 \leqslant i < k}, \\ (\bar{l}_i, u_i \ominus m)_{1 \leqslant i \leqslant k}, \\ (\square, m) \end{array} \right\},$$

where $m = \min\{w, u_1, u_2, \ldots, u_k\}$.

This rule can be proved in k resolution steps. Assume, without loss of generality, that $\forall_{1 \leqslant i < k} \, u_i \leqslant u_{i+1}$. Assume as well that $u_k < \top$ (otherwise unit clause reduction could have been previously triggered). Let $m = \min\{w, u_1\}$. Fig. 7 shows the corresponding resolution tree. Recall that bold clauses are resolved, so they must be removed from the formula. Essentially, each unit clause is used to eliminate one literal from the length k clause. At the end, we derive the empty clause.

5.3.2. Dominating unit-clause

The *dominating unit-clause* rule [9] (also called UP3 in [13]) says that if the weight of a unit clause (l, u) is higher than the sum of weights in which $\bar{l}$ appears, we can safely assign l to the formula. Formally,

$$\mathcal{F} = \left\{ (l, u) \right\} \cup \left\{ (l \vee A_i, u_i) \right\}_{i=1}^{k'} \cup \left\{ (\bar{l} \vee B_j, w_j) \right\}_{j=1}^{k} \cup \mathcal{F}'$$

with $u \geqslant \sum_{j=1}^{k} w_j$ and $\mathcal{F}'$ does not contain any occurrence of l or $\bar{l}$, then

$$\mathcal{F} \equiv \left\{ (B_j, w_j) \right\}_{j=1}^{k} \cup \mathcal{F}'.$$

This rule can be proved in k resolution steps plus the application of the pure literal rule. Fig. 8 shows the corresponding resolution tree. As in the previous case, we can assume that weight u is less than $\top$ because otherwise the unit clause
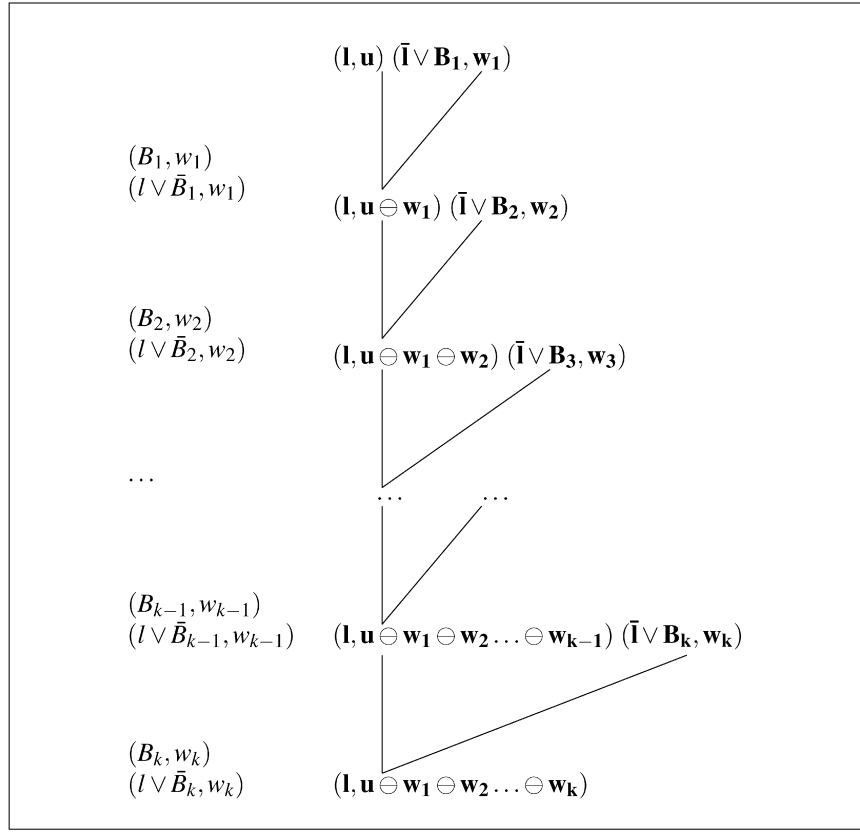
Fig. 8. Resolution tree for the *dominating unit clause rule*.

reduction could have been triggered. At each step, unit clause l is resolved with one $(\bar{l} \vee B_j, w_j)$. Since, by definition, the weight of l is larger than or equal to w_j, clause $\bar{l} \vee B_j$ is replaced by B_j. At the end of the process, there is no clause mentioning $\bar{l}$, so the pure literal rule can be applied, which proves the correctness of the rule.
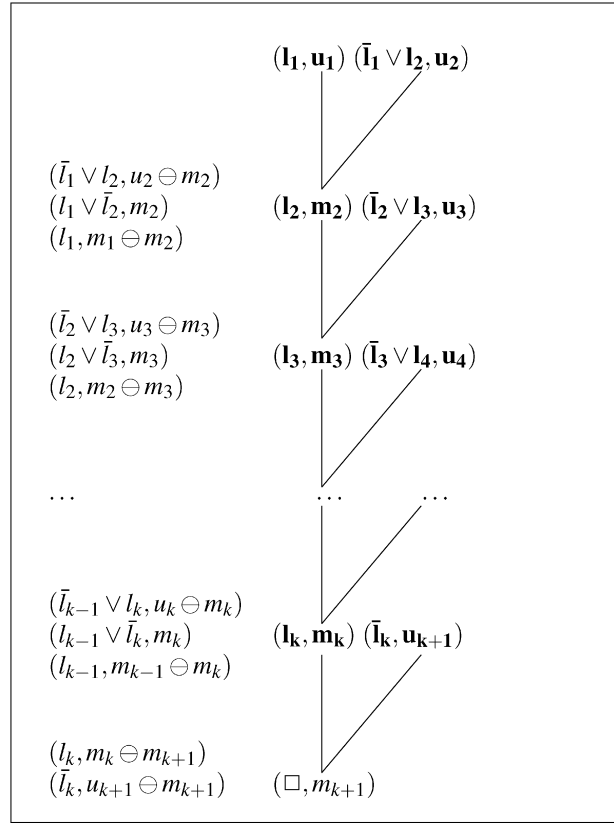
5.3.3. Chain resolution

Our original *chain resolution* rule identifies a subset of *chained* binary clauses and two unit clauses associated to the ends. When such pattern exists, a sequence of unit resolution steps suffices to derive the empty clause. The rule is the following,

$$\left\{ \begin{array}{l} (l_1, u_1), \\ (\bar{l}_i \vee l_{i+1}, u_{i+1})_{1 \leqslant i < k}, \\ (\bar{l}_k, u_{k+1}) \end{array} \right\} \equiv \left\{ \begin{array}{l} (l_i, m_i \ominus m_{i+1})_{1 \leqslant i \leqslant k}, \\ (\bar{l}_i \vee l_{i+1}, u_{i+1} \ominus m_{i+1})_{1 \leqslant i < k}, \\ (l_i \vee \bar{l}_{i+1}, m_{i+1})_{1 \leqslant i < k}, \\ (\bar{l}_k, u_{k+1} \ominus m_{k+1}), \\ (\square, m_{k+1}) \end{array} \right\},$$

where $m_i = \min\{u_1, u_2, \ldots, u_i\}$ and $\forall_{1 \leqslant i < j \leqslant k} \; var(l_i) \neq var(l_j)$. This rule can also be proved in k steps of resolution. Fig. 9 shows the corresponding resolution tree. Starting with unit clause l_1, at each resolution step a unit clause l_i is resolved with $(\bar{l}_i \vee l_{i+1}, u_{i+1})$, which produces the unit clause l_{i+1} to be used in the following resolution step. The last unit clause obtained is l_k and it is resolved with $(\bar{l}_k, u_{k+1})$, which derives the empty clause.

Example 31. Consider the following formula $\{(x, 2), (\bar{x} \vee y, 1), (\bar{y} \vee z, \top), (\bar{z}, 2)\}$. If we resolve $(x, 2)$ and $(\bar{x} \vee y, 1)$ we obtain $\{(x, 1), (y, 1), (x \vee \bar{y}, 1), (\bar{y} \vee z, \top), (\bar{z}, 2)\}$. If we resolve $(y, 1)$ and $(\bar{y} \vee z, \top)$ we obtain $\{(x, 1), (x \vee \bar{y}, 1), (z, 1), (y \vee \bar{z}, 1), (\bar{y} \vee z, \top), (\bar{z}, 2)\}$. Next, if we resolve $(z, 1)$ and $(\bar{z}, 2)$, we obtain $\{(x, 1), (x \vee \bar{y}, 1), (y \vee \bar{z}, 1), (\bar{y} \vee z, \top), (\bar{z}, 1), (\square, 1)\}$.

Observe that chain resolution with $k = 1$ reduces to simple neighborhood resolution, with $k = 2$ reduces to the *star rule* limited to binary clauses, with $k = 3$, it is the 3-RES rule proposed in [2].

Fig. 9. Resolution tree for the *chain resolution rule*.

5.3.4. Cycle resolution

Our original *cycle resolution* rule identifies a subset of binary clauses with a cyclic structure. When such a pattern exists, a sequence of resolution steps with binary clauses suffices to derive a new unit clause. The rule is the following,

$$
\left\{
\begin{array}{l}
(\bar{l}_i \vee l_{i+1}, u_i)_{1 \leqslant i < k}, \\
(\bar{l}_1 \vee \bar{l}_k, u_k)
\end{array}
\right\}
\equiv
\left\{
\begin{array}{l}
(\bar{l}_1 \vee l_i, m_{i-1} \ominus m_i)_{2 \leqslant i \leqslant k}, \\
(\bar{l}_i \vee l_{i+1}, u_i \ominus m_i)_{2 \leqslant i < k}, \\
(\bar{l}_1 \vee l_i \vee l_{i+1}, m_i)_{2 \leqslant i < k}, \\
(l_1 \vee \bar{l}_i \vee l_{i+1}, m_i)_{2 \leqslant i < k}, \\
(\bar{l}_1 \vee \bar{l}_k, u_k \ominus m_k), \\
(\bar{l}_1, m_k)
\end{array}
\right\},
$$

where $m_i = \min\{u_1, u_2, \ldots, u_i\}$ and $\forall_{1 \leqslant i < j \leqslant k} \, var(l_i) \neq var(l_j)$. This rule can be proved in $k - 1$ steps of resolution. Fig. 10 shows the corresponding resolution tree. The use of the cycle rule is to derive new unit clauses that, in turn, can be used by chain resolution to increase the weight of the empty clause.

Example 32. Consider the formula $\{(x_1 \vee x_2, 1), (\bar{x}_1 \vee x_3, 1), (\bar{x}_2 \vee x_3, 1), (\bar{x}_3 \vee \bar{x}_4, 1), (x_4 \vee x_5, 1), (\bar{x}_5, 1)\}$. We can apply the cycle rule to the three first clauses obtaining, $\{(x_3, 1), (x_1 \vee x_2 \vee \bar{x}_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee x_3, 1), (\bar{x}_3 \vee \bar{x}_4, 1), (x_4 \vee x_5, 1), (\bar{x}_5, 1)\}$. Chain resolution can be applied to the unary and binary clauses producing, $\{(x_1 \vee x_2 \vee \bar{x}_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee x_3, 1), (x_3 \vee x_4, 1), (\bar{x}_4 \vee \bar{x}_5, 1), (\Box, 1)\}$.

6. An efficient Max-SAT solver

In the previous section we presented a set of simplification rules. Some of them have been previously proposed by other researchers, while some others are original. We showed that all of them can be viewed as special cases of resolution, hyper-resolution or variable elimination. In this section we consider their incorporation into the Max-DPLL algorithm introduced in Section 4.2. The idea is to use these rules to simplify the current Max-SAT formula before
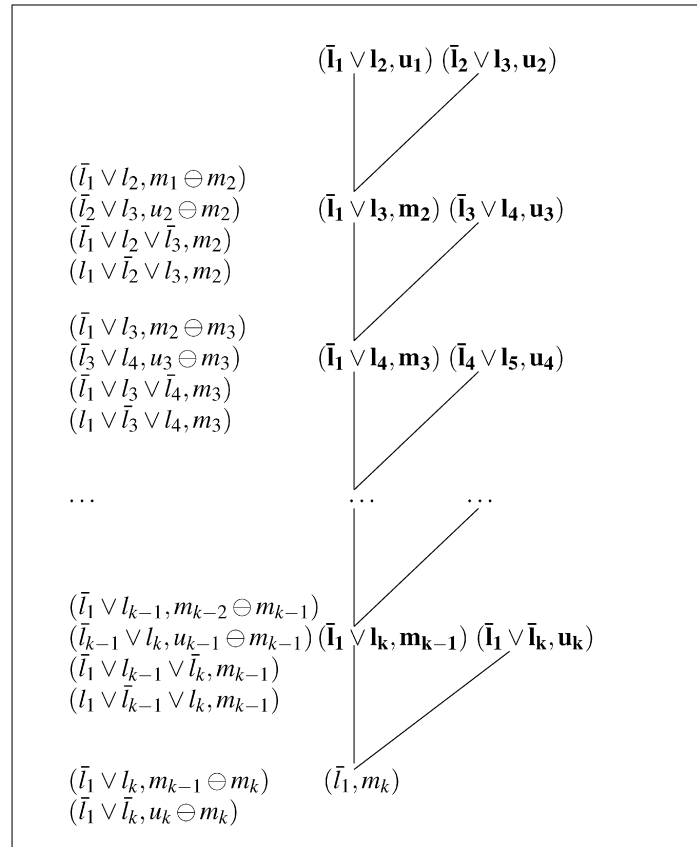
Fig. 10. Resolution tree for the *cycle resolution rule*.

letting Max-DPLL branch on one of the variables. Our experimental work indicates that it is not cost effective to apply all of them on a general basis. We have observed that only three rules are useful in general: *neighborhood resolution*, *chain resolution* and *cycle resolution*. Besides, it only pays off to apply these rules to clauses of very small size (up to 2). The reason being that there is only a quadratic number of them which bounds the overhead of the detection of situations when they can be triggered. Regarding cycle resolution, we only found effective to apply the $k = 3$ case (namely, considering triplets of variables). Note that the fact that our solver only incorporates these three rules, does not prevent other rules from being effective in a particular class of problems where we did not experiment.

A high-level description of our solver appears in Fig. 11. It is Max-DPLL augmented with the simplification rules in function `Simplify`. This function iteratively simplifies the formula. It stops when an explicit contradiction is derived or no further simplification can be done (line 10). Simplification rules are arranged in an ordered manner, which means that if two rules R and R' can be applied, and rule R has higher priority than rule R', the algorithm will chose R. The rules with the highest priority are *unit clause reduction* and *absorption* grouped in the assignment $\mathcal{F}[l]$ operation (line 3). Next, we have *aggregation* (line 4), *hardening* (line 5), *neighborhood resolution* (line 6), *chain resolution* (line 7) and *cycle resolution* restricted to cycles of length 3 (line 8). All the rules are restricted to unary and binary clauses. We do not apply any variable elimination rule.

Although our actual implementation is conceptually equivalent to the pseudo-code of Fig. 11 it should be noted that such code aims at clarity and simplicity. Thus, a direct translation into a programming language is highly inefficient. The main source of inefficiency is the time spent searching for clauses that match with the left-hand side of the simplification rules. This overhead, which depends on the number of clauses, takes place at each iteration of the loop. As we mentioned, our current implementation only takes into account clauses of arity less than or equal to two. Another way to decrease such overhead is to identify those events that potentially make a transformation applicable. For instance, a clause may be made mandatory (line 5) only when its weight or the weight of the empty clause increases. Then, our implementation reacts to these events and triggers the corresponding rules. Such approach is well-known in the constraint satisfaction field and it is usually implemented with streams of pending events [31,32].

```
function Simplify(F, ⊤)
1.  stop := false
2.  do
3.      if (l, ⊤) ∈ F then apply F[l]
4.      elseif {(C, u), (C, w)} ⊆ F then apply Aggregation
5.      elseif {(□, u), (C, w)} ⊆ F ∧ u ⊕ w = ⊤ then apply Hardening
6.      elseif {(x ∨ A, u), (x̄ ∨ A, w)} ⊆ F then apply Neighborhood Res.
7.      elseif {(l₁, u₁), (l̄ᵢ ∨ lᵢ₊₁, uᵢ₊₁)₁≤ᵢ<ₖ, (l̄ₖ, uₖ₊₁)} ⊆ F then apply Chain Res.
8.      elseif {{(l ∨ h, u), (l̄ ∨ q, v), (h̄ ∨ q, w)}} ⊆ F then apply Cycle Res.
9.      else stop := true
10. until (((□, ⊤) ∈ F) ∨ stop)
11. return (F)
endfunction
function Max-DPLL(F, ⊤) return ℕ
12. F := Simplify(F, ⊤)
13. if F = ∅ then return 0
14. if F = {(□, w)} then return w
15. l := SelectLiteral(F)
16. ⊤ := Max-DPLL(F[l], ⊤)
17. ⊤ := Max-DPLL(F[l̄], ⊤)
18. return ⊤
endfunction
```

Fig. 11. Max-DPLL enhanced with inference. Function Simplify$(\mathcal{F}, \top)$ converts the input formula into a simpler one. Note that in our implementation, for efficiency reasons, we only consider the $|A| \leq 1$ and $|C| \leq 2$ case.

The way in which we detect the chain resolution pattern also deserves special consideration. At each search node, we consider the set of binary and unary clauses and compute the corresponding implication graph defined as follows:

- for each variable x_i, the graph has two vertices x_i and $\bar{x}_i$,
- for each binary clause $(l_i \vee l_j, u)$, the graph has two arcs: $(\bar{l}_i, l_j)$ and $(\bar{l}_j, l_i)$. We say that these two arcs are *complementary*,
- if the formula contains the unit clause (l, u), we say that vertex l is a *starting* vertex, and vertex $\bar{l}$ is an *ending* vertex.

It is easy to see that if there is a path $(l_1, l_2, \ldots, l_k)$, where l_1 and l_k are starting and ending vertices, respectively, and the path does not cross any pair of complementary arcs, then chain resolution can be applied and the path tells the order in which resolution must be applied.

In our implementation, we select one arbitrary starting vertex and compute shortest paths to all ending vertices using Dijkstra's algorithm. If one of the paths does not cross complementary arcs, we trigger the rule. Else, another starting vertex is selected and the process is repeated. Note that this method does not necessarily detect all the potential applications of chain resolution because it only takes into consideration one path between each pair of starting and ending vertices (the shortest path given by Dijkstra). The fact that this path crosses complementary arcs does not prevent the existence of other paths that do not cross complementary arcs. We believe that a better approach would be to use a flow algorithm, but we have not yet studied this possibility.

7. Experimental results

We divide the experiments in two parts. The purpose of the first part is to assess the importance of the different inference rules that our solver incorporates. These experiments include random Max-SAT instances and random Max-Clique problems. The purpose of the second part is to evaluate the performance of our solver in comparison to other available solving techniques. These experiments include random weighted and unweighted Max-SAT instances, random and structured Max-One problems, random Max-Cut problems, random, structured and real Max-Clique problems and combinatorial auctions.

Our solver, written in C, is available as part of the TOOLBAR software[1] (version 3.0). Benchmarks are also available in the TOOLBAR repository. In all the experiments with random instances, samples have 30 instances and plots report mean CPU time in seconds. Executions were made on a 3.2 GHz Pentium 4 computer with Linux. Unless otherwise indicated, executions were aborted when they reached a time limit of 1200 seconds. In all the plots' legend, the order of the items reflects the relative performance order of the different competitors. Default options were used for all the competitors' solvers. No initial upper bounds were given, so the reported CPU time is the time to find an optimal solution and prove its optimality.

7.1. Adding inference to Max-DPLL

We consider the following versions of our solver:

(1) Basic Max-DPLL. Namely, Algorithm 11 in which lines 6–8 in Function `Simplify` are commented out. We denote this algorithm Max-DPLL-1.
(2) The previous algorithm enhanced with *neighborhood resolution* (namely, lines 7–8 in `Simplify` are commented out). We denote this algorithm Max-DPLL-2.
(3) The previous algorithm enhanced with *chain resolution* (namely, line 8 in `Simplify` is commented out). We denote this algorithm Max-DPLL-3.
(4) The previous algorithm enhanced with *cycle resolution* (namely, all the lines in `Simplify` are considered). We denote this algorithm Max-DPLL-4.

For the first experiment we consider random Max-k-SAT instances. A *random k-SAT formula* is defined by three parameters $\langle k, n, m \rangle$. k is the fixed size of the clauses, n is the number of variables and m is the number of clauses. Each clause is randomly generated by selecting k distinct variables with a uniform probability distribution. The sign of each variable in each clause is randomly chosen. In the following experiments we generate instances in which the number of clauses is always sufficiently high as to make the formula unsatisfiable and we solved the corresponding Max-SAT problem. We used the *Cnfgen*[2] generator. Note that it allows repeated clauses, so v repetitions of a clause C are grouped into one weighted clause (C, v).

Fig. 12 (top-left) reports the results on random Max-2-SAT instances with 100 variables and varying number of clauses. It can be seen that Max-DPLL-1 performs very poorly and can only solve instances with up to 200 clauses. The addition of neighborhood resolution (namely, Max-DPLL-2) improves its performance by 2 orders of magnitude and allows to solve instances with up to 300 clauses. The further addition of chain resolution provides a spectacular improvement which allows to solve instances with up to 750 clauses. Finally, the addition of cycle resolution allows to solve in 100 seconds instances of up to 1000 clauses. The same improvements are observed on random Max-3-SAT instances (Fig. 12 top-right).

The Max-Clique problem is the problem of finding a clique of maximum size embedded in a given graph. It is known that solving the maximum clique problem of a graph $G = (V, E)$ is equivalent to solving the minimum vertex covering problem of graph $G' = (V, E')$ where E' is the complementary of E (namely, $(u, v) \in E'$ if and only if $(u, v) \notin E$). Therefore, we solved Max-Clique instances by encoding into Max-SAT the corresponding minimum vertex covering problem, as described in Example 5 in Section 3.

A *random graph* is defined by two parameters $\langle n, e \rangle$ where n is the number of nodes and e is the number of edges. Edges are randomly chosen using a uniform probability distribution. Fig. 12 (bottom) reports the results of solving the Max-Clique problem of random graphs with 150 nodes and varying number of edges. It can be observed that the instances with connectivity lower than 50 percent are trivially solved by the four algorithms. Note that instances with small connectivity have an associated Max-SAT encoding containing a large number of hard clauses. Hence, the *unit clause reduction rule* is applied very frequently on those instances. This is the reason why they are so easily solved. However, as the connectivity increases, the differences between all the versions also increases. We noticed a slight improvement for Max-DPLL-2 over Max-DPLL-1. For connectivities between 76% and 99% the greatest differences
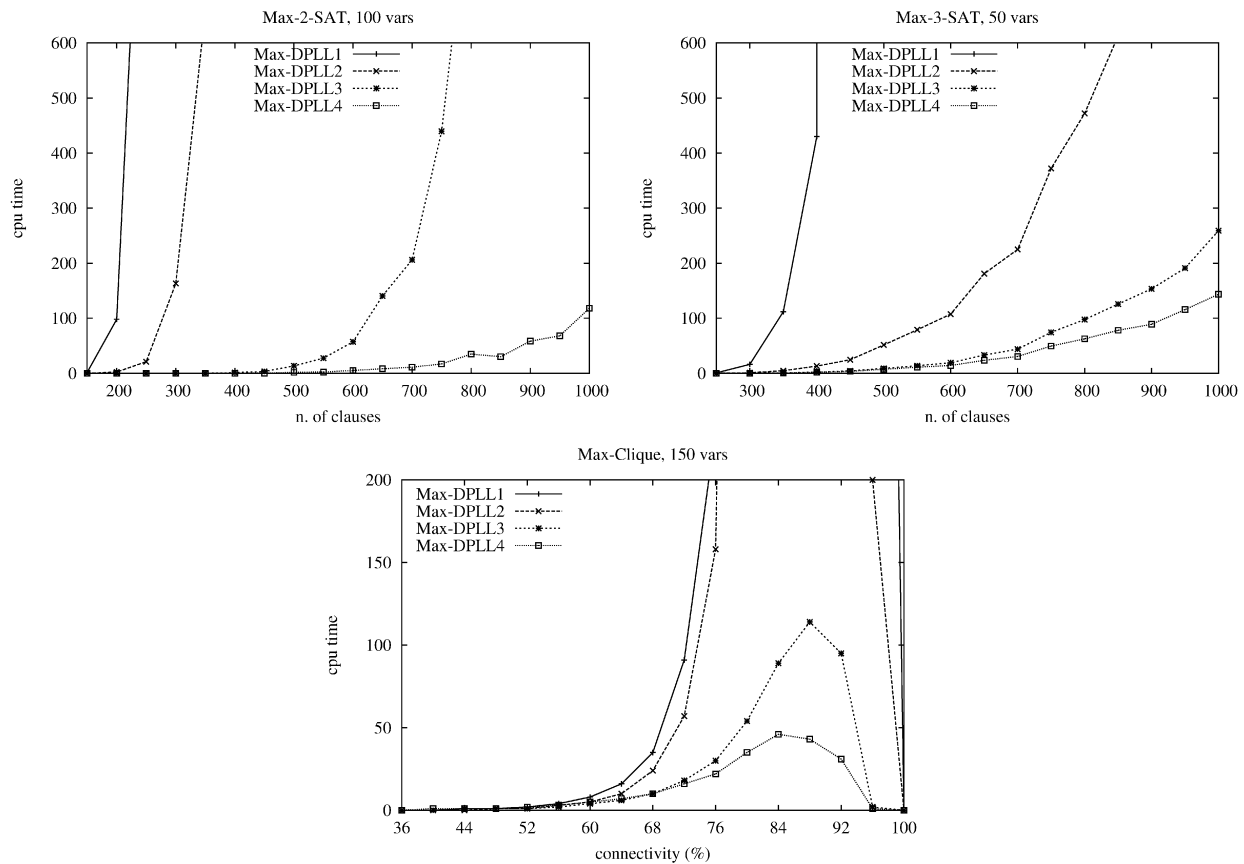

Fig. 12. Experimental results of different algorithms on random Max-*k*-SAT and Max-Clique instances.

are found. While Max-DPLL-1 and Max-DPLL-2 are unable to solve those instances, both Max-DPLL-3 and Max-DPLL-4 perform well. With a connectivity near to 90%, it can be observed that using the cycle resolution reports a noticeable improvement.

From these experiments we conclude that the synergy of the three inference rules of Max-DPLL-4 produces an efficient algorithm.

7.2. Max-DPLL versus alternative solvers

In the following experiments, we evaluate the performance of Max-DPLL-4 (we will refer to it as MAX-DPLL). For that purpose, we compare MAX-DPLL with the following state-of-the-art Max-SAT solvers: MAXSOLVER (September 2004 second release) [13], LAZY (version 2.0) [14], UP (version 1.5) [33], LB4A [12] and MAXSATZ (*July 2006* release) [34]. They suffer from the following limitations:

- The available version of MAXSOLVER is restricted to instances with less than 200 variables and 1000 clauses.
- For implementation reasons, UP and MAXSATZ cannot deal with instances having clauses with high weights. Therefore, they cannot deal with instances that combine mandatory and weighted clauses.
- LB4A can only solve unweighted Max-2-SAT problems (i.e., it is restricted to binary clauses with unit weights and without repeated clauses).

Consequently, in the experiments, we will only execute a solver if it is possible, according to its limitations.

It is known that Max-SAT problems can also be solved with *pseudo-Boolean* and SAT solvers. For the sake of a more comprehensive comparison, we also consider PUEBLO (version 1.4) [35] and MINISAT+ (2005 release) [36], which are among the best pseudo-Boolean and SAT solvers, respectively. In Appendix B, we describe how we translated the Max-SAT instances into these two frameworks. Note that pseudo-Boolean formulas are equivalent to 0–1
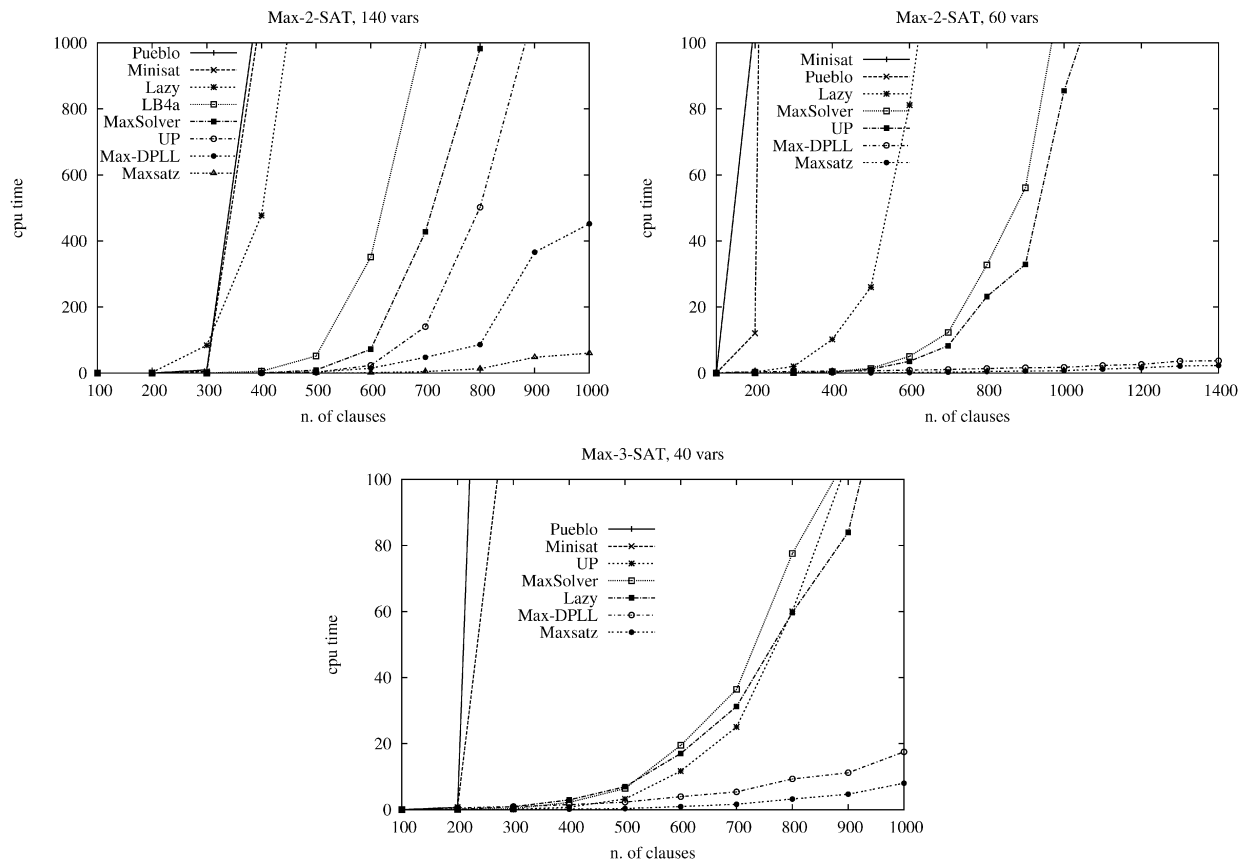
Fig. 13. Random Max-2-SAT and Max-3-SAT. Max-2-SAT instances on the plot on the left do not contain repeated clauses.

integer linear programs (ILP). Thus, they can also be solved with a state-of-the-art ILP solver such as CPLEX. We have not considered this alternative because [11] showed that it is generally ineffective for Max-SAT instances. Max-SAT problems can also be solved with WCSP solvers [11]. We have not consider this type of solver in our study, because the reference WCSP solver is MEDAC [32], which uses techniques similar to those of Max-DPLL and can be roughly described as a non-Boolean restricted version of Max-DPLL-3.

7.2.1. Random Max-k-SAT

For the following experiment, we generated random 2-SAT instances of 60 variables and 3-SAT instances of 40 variables with varying number of clauses using the *Cnfgen* generator. We also generated random 2-SAT instances of 140 variables using the 2-SAT generator of [12] that does not allow repeated clauses.

Fig. 13 (top-left) presents the results on Max-2-SAT without repeated clauses. It can be observed that MAX-DPLL and MAXSATZ are the only algorithms that can solve problems of up to 1000 clauses. MAXSATZ is roughly 3 times faster than MAX-DPLL. A surprising observation is that the LB4A solver, which was specifically designed for Max-2-SAT without repetitions, performs worse than the other Max-SAT solvers in random unweighted Max-2-SAT. Fig. 13 (top-right) presents the results on Max-2-SAT with repeated clauses. MAX-DPLL and MAXSATZ are the best algorithms, with similar performance. The third best solver, UP, is nearly 100 times slower in the hardest instances. Fig. 13 (bottom) presents the results on Max-3-SAT. MAXSATZ provides the best performance. The second best option MAX-DPLL requires twice as much time. The third best option LAZY is about 20 times slower than MAXSATZ. An observation worth noting is that the alternative encodings (namely, pseudo-Boolean and SAT) do not seem to be effective in these random instances.

We can conclude from this experiment that MAXSATZ is the best solver for this type of problems (random and unweighted). The second best option is MAX-DPLL. We explain in Section 8 that MAXSATZ can be roughly described as UP enhanced with inference rules similar to those used by MAX-DPLL. Therefore, the importance of inference rules based on weighted resolution is corroborated.
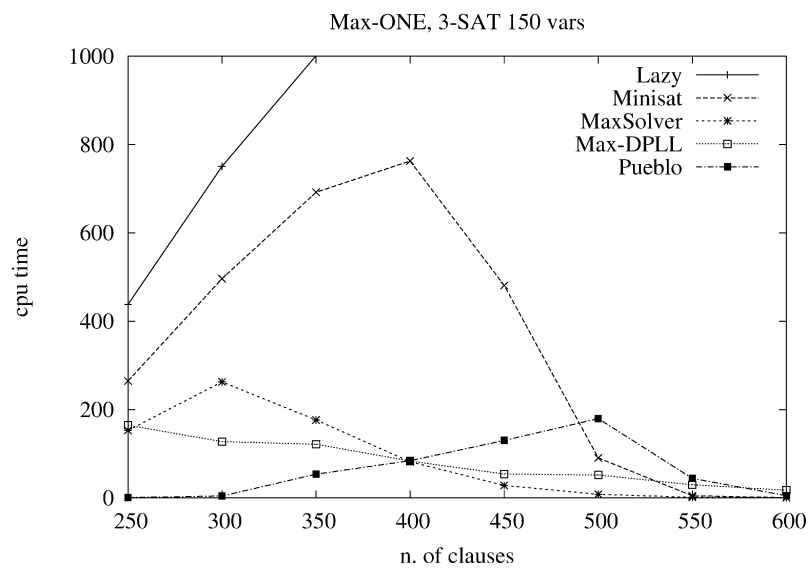
Max-ONE, 3-SAT 150 vars

Fig. 14. Random Max-One instances.

7.2.2. 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.

Fig. 14 shows the results with random 3-SAT instances of 150 variables. Note that MAXSATZ cannot be executed in this benchmark because it cannot deal with mandatory clauses. The first thing to be observed is that LAZY and MINISAT do not perform well. Regarding the other solvers, PUEBLO is the best when the number of clauses is very small, but its relative efficiency decreases as the number of clauses grows. MAXSOLVER has the opposite behavior, and MAX-DPLL always lay in the middle. The performance of all these solvers converges as the number of clauses approaches the phase transition peak. The reason is that, as the number of models decreases, the optimization part of the Max-One problem loses relevance (the number of models to chose from decreases).

Fig. 15 reports results on the Max-One problem on selected satisfiable SAT instances from the DIMACS challenge [37].[3] The first column indicates the name of the problem classes. The second column indicates the number of instances of each class. The other columns indicate the performance of each solver by indicating the number of instances that could be solved within the time limit. If all the instances could be solved, the number in parenthesis is the mean time in seconds. The "–" symbol in the MAXSOLVER column indicates that the instances could not be executed due to the limitation that this solver has on the maximum number of variables and clauses. As can be observed, MAX-SOLVER and LAZY do not succeed in this benchmark, which means that MAX-DPLL is the only Max-SAT solver that can deal with it. Its performance is comparable to the good performance of MINISAT and PUEBLO. However, in the *par16*c** instances MAX-DPLL performs badly, while in the *par8** instances it performs better than the others.

7.2.3. 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 in finding a cut of maximum size. It is encoded as Max-SAT associating one variable x_i to each graph vertex. Value **t** (respectively, **f**) indicates that vertex v_i belongs to U (respectively, to $V - U$). For each edge (v_i, v_j), there are two clauses $x_i \vee x_j$ and $\bar{x}_i \vee \bar{x}_j$. Given a complete assignment, the number of falsified clauses is $|E| - S$ where S is the size of the cut associated to the assignment. Note that this encoding produces an unweighted Max-2-SAT formula, so the LB4A solver can be used.

---

[3] ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf.

Problem	n. inst.	MaxDPLL	MaxSolver	Lazy	MiniSat	Pueblo
aim50*	16	16(0.59)	16(0.12)	16(28.25)	16(0.01)	16(0.00)
aim100*	16	16(2.67)	16(4.92)	0	16(0.02)	16(0.00)
aim200*	16	9	4	0	16(0.03)	16(0.00)
jnh*	16	16(1.49)	–	6	16(0.08)	16(0.10)
ii8*	14	5	–	1	10	3
ii32*	17	11	–	0	16	15
par8*	10	10(0.92)	–	5	10(16.39)	10(26.52)
par16*c*	5	5(784.14)	–	0	5(0.93)	5(0.93)

Fig. 15. Results for the Max-One problem on selected DIMACS SAT instances.
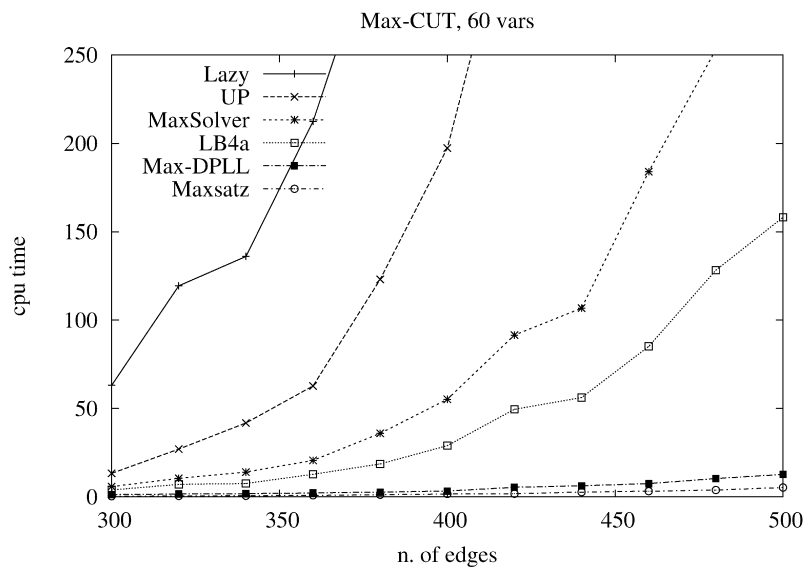

Fig. 16. Random Max-Cut instances.

Random Max-Cut instances are extracted from random graphs (see Section 7.1 for their generation). We considered graphs of 60 nodes and varying number of edges.

Fig. 16 reports the results on this benchmark. It can be observed that for all the solvers except for MAXSATZ and MAX-DPLL, problems become harder as the number of edges increases. However, MAXSATZ and MAX-DPLL solve instances of up to 500 edges almost instantly. The third best solver is LB4A, but MAXSATZ is up to 30 times faster. PUEBLO and MINISAT perform so poorly even in the easiest instances that they are not included in the comparison.

7.2.4. Max-Clique

The Max-Clique problem is the problem of finding a clique of maximum size embedded in a given graph. Its Max-SAT encoding was described in Section 7.1. MAXSATZ, UP, MAXSOLVER and LB4A solvers could not be executed in this domain due to their limitations. Our first Max-Clique experiment used random graphs with 150 nodes and varying number of edges. Fig. 17 reports the results. Again, MAX-DPLL is clearly better than any other competitor. All other competitors are more than 2 orders of magnitude slower than MAX-DPLL.

We also considered the 66 Max-Clique instances from the DIMACS challenge [37].[4] MAXSOLVER could not be executed in this benchmark because the number of variables and clauses of the instances exceed its capacity. Thus, the only two Max-SAT solvers that could be executed are MAX-DPLL and LAZY. Within the time limit, they solved 32 and 23 instances, respectively. MINISAT and PUEBLO could solve 22 and 16 instances, respectively. Therefore, MAX-DPLL provided the best performance in this benchmark, too.
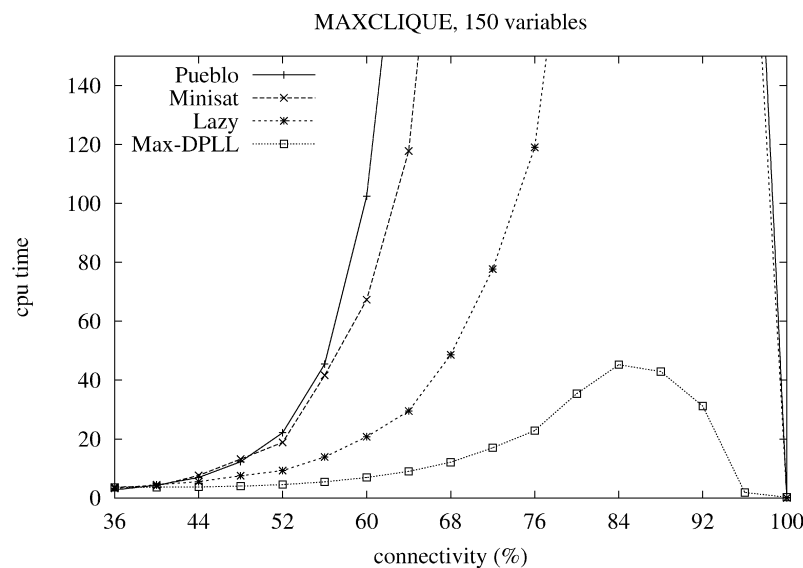
---

[4] ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique.

MAXCLIQUE, 150 variables

Fig. 17. Random Max-Clique instances.

Problem	$|V|$	$|C|$	Opt	MaxDPLL	Lazy	MiniSat	[4]
1bpi–1knt	2,279	2,213,051	31	3,854	–	25,303	19
1bpi–2knt	2,436	2,521,619	29	6,891	–	11,502	182
1bpi–5pti	3,016	3,851,441	42	–	–	24,607	30
1knt–1bpi	2,494	2,649,173	30	6,601	–	19,310	110
1knt–2knt	1,806	1,391,200	39	904	12,379	5,710	0
1knt–5pti	2,236	2,122,357	28	4,749	–	31,535	46
1vii–1cph	171	13,125	6	0.27	0.32	0.46	0
2knt–5pti	2,184	2,021,705	28	3,911	–	10,239	95
3ebx–1era	2,548	2,769,706	31	9,332	–	52,434	236
3ebx–6ebx	1,768	1,338,035	28	1,636	16,581	19,482	6
6ebx–1era	1,666	1,189,537	20	1,428	14,651	19,367	101

Fig. 18. Protein structure alignment problem transformed into Max-Clique.

These instances have been previously used to evaluate several dedicated maximum clique algorithms. Performing a proper comparison with MAX-DPLL is difficult because their codes are not available and we would need to re-program their algorithms. However, following the approach of [38], we overcome this problem by normalizing the reported times. Of course, this is a very simplistic approach which disregards very relevant parameters such as the amount of memory or the processor model. In consequence, the following results can only be taken as indicative information. Giving a time limit of 2.5 hours per instance on a 3.2 GHz computer, MAX-DPLL was able to solve 37 instances. In an *equivalent* (via normalization) time, [39] solves 38, [40] solves 36, [41] solves 45, and [38] solves 52.

Finally, we considered 11 Max-Clique real instances, provided by J.S. Sokol, corresponding to the protein structure alignment problem transformed into the maximum clique problem as described in [4] (instances are taken before the preprocessing described in the paper). In this problem, the goal is to compute a score of similarity between two proteins based on a particular knowledge of their respective three-dimensional structure. Due to the size of the instances and the presence of mandatory and weighted clauses, only three solvers could be executed, MAX-DPLL, LAZY and MINISAT. The results are shown in Fig. 18 which gives the instance name as used in [4], the number of Boolean variables, the number of clauses, the maximum clique size, the CPU time (in seconds, with a time limit of 10 hours) of MAX-DPLL (run on a 3 GHz Intel Xeon 64-bit with 16 GB), of LAZY and MINISAT (run on a 3 GHz Intel Xeon with 4 GB), and of the specialized algorithm introduced in [4] (run on a 200 MHz SGI):
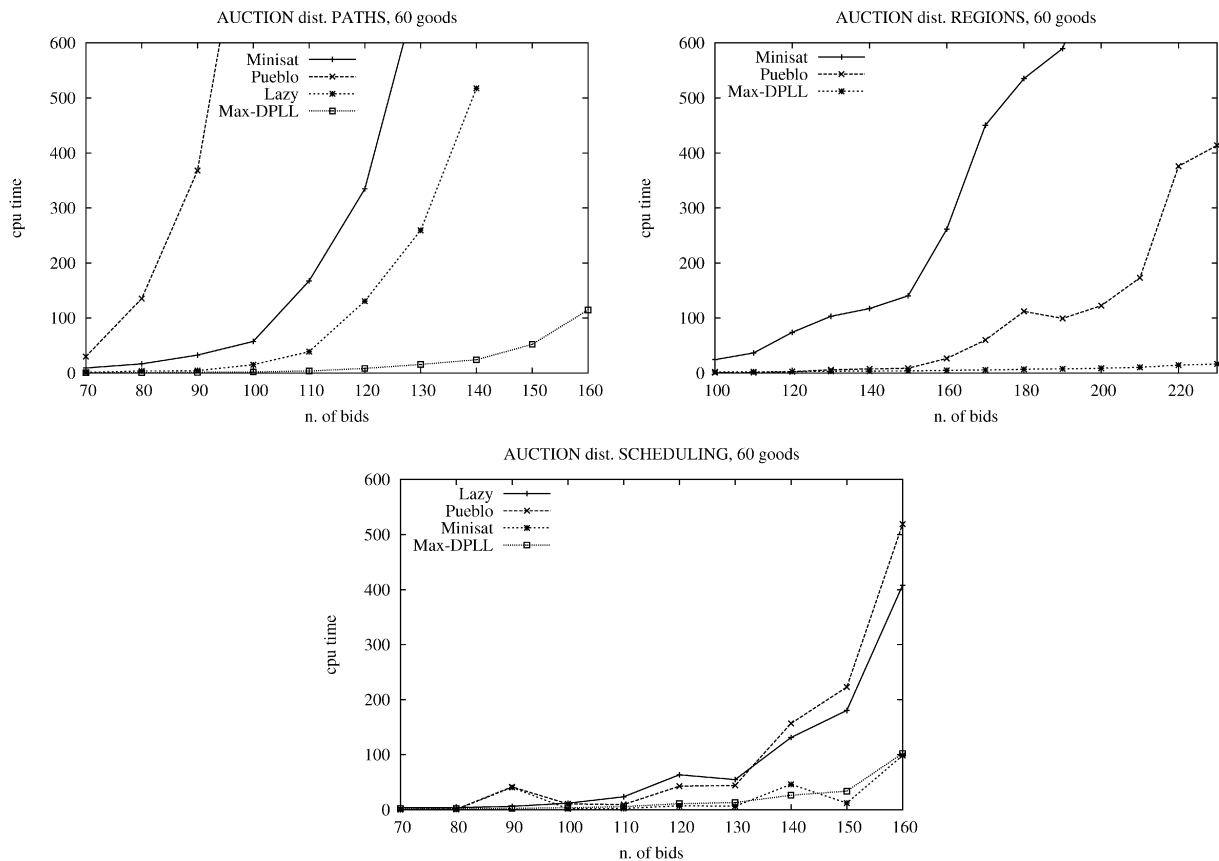
Fig. 19. Combinatorial auctions. Top-left: *Paths* distribution. Top-right: *Regions* distribution. Bottom: *Scheduling* distribution.

LAZY and MINISAT require less memory than MAX-DPLL which could not solve 1bpi–5pti for memory reasons. LAZY could solve 4 instances only in less than 10 hours. Although MINISAT solves all the instances, it is approximately 5 times slower than MAX-DPLL on 10 instances. The dedicated algorithm [4] solves all the instances several orders of magnitude faster than MAX-DPLL, showing the gap between generic and specialized algorithms.

7.2.5. Combinatorial auctions

Combinatorial auction allows bidders to bid for indivisible subsets of goods. Consider a set G of goods and n bids. 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 [7]. 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, we add a mandatory clause $(\bar{x_i} \vee \bar{x_j}, \top)$.

We used the CATS generator [42] 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 varied 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. MAXSATZ, UP, MAXSOLVER and LB4A could not be executed due to their limitations. The LAZY solver could not be included in the *Regions* comparison due to overflow problems.

Fig. 19 (top-left) presents the results for the Paths distribution. MAX-DPLL produces the best results being 22 times faster than the second best option LAZY. Fig. 19 (top-right) presents the results for the Regions distribution. MAX-DPLL is again the best algorithm. It is 26 times faster than the second best solver PUEBLO. Finally, results for the Scheduling distribution are shown in Fig. 19 (bottom). In this benchmark, the performance of MAX-DPLL and MINISAT are quite similar, while the other solvers are up to 4 times slower.

8. Related work

8.1. Relation with weighted constraint satisfaction problems

A weighted constraint satisfaction problem (WCSP) [43] is similar to a Max-SAT instance except that: variables are multi-valued rather than bi-valued, and costs are given by arbitrary cost functions, rather than by clauses. It is known that a WCSP can be translated into a Max-SAT instance and vice-versa [11].

Some of the ideas presented in this paper have strong connections to different techniques recently developed in the WCSP field. In the following we summarize these connections. The idea of adding a lower bound $(\Box, w)$ and an upper bound $\top$ to the problem formulation has been borrowed from [44]. The concept of equivalence-preserving problem transformation by moving costs comes from [26]. DP and Max-DP can be seen as instantiations of *bucket elimination* [45], a meta-algorithm based on the variable elimination principle, which is well-known in the WCSP context [46,47]. *Neighborhood resolution* is related to the notion of *projection* and has been used to enforce *node* and *arc-consistency* [26,48]. The application of *Chain resolution* with $k = 2$ is similar to the enforcement of *existential arc consistency* [32]. Cycle resolution with $k = 3$ is related to one particular case of the so-called *high-order* consistencies proposed in [49]. More precisely, it is a weighted version restricted to Boolean variables of *path inverse consistency* [50].

8.2. Relation with other Max-SAT solvers

In the last years several algorithms for Max-SAT have been proposed [11–15,30,33,51]. Most of them have in common a basic branch and bound structure and mainly differ in the lower bound that they use. When the lower bound does not reach the upper bound, search continues below the current node and new lower bounds must be computed. The problem with all these lower bound computation methods is that they do not transform the formula and make the lower bound explicit as part of the problem. Consequently, the lower bound needs to be computed from scratch at each visited node and the same inconsistencies may be *discovered* again and again. In our approach the lower bound is computed as part of a simplification process and becomes part of the current problem. When the lower bound does not reach the upper bound, the simplified formula is passed to descendant nodes, so they actually inherit the lower bound. Interestingly, Li et al. have also detected this common pitfall. In their last work [34], done independently and in parallel to ours, they introduce a Max-SAT solver with many similarities to our Max-DPLL. The main contribution of their paper are 6 Max-SAT simplification rules (called Rules 1–6). They conclude, as we do in this paper, that there is a small set of simplification rules that seems to be essential for the efficiency of a Max-SAT solver. Besides, their set of *fundamental* simplification rules is very similar to ours. Their solver, called MAXSATZ, is build on top of their previous solver [33,51]. The novelty of MAXSATZ is that it incorporates *neighborhood resolution* restricted to unary and binary clauses (Rules 1 and 2 using their terminology), *chain resolution* (Rules 3 and 4), a special case of *cycle resolution* restricted to triplets of variables in which an immediate lower bound increment is guaranteed (Rule 5) and, finally, a sequence of *chain resolution* followed by *cycle resolution* in which, again, a lower bound increment is guaranteed (Rule 6). Besides, MAXSATZ identifies that a certain variable must necessarily take one of its two truth values exactly as our *hardening* rule does.

A significant difference between MAXSATZ and Max-DPLL is the type of Max-SAT instances that are assumed to be solved. MAXSATZ assumes unweighted clauses or weighted clauses with low weights. As a consequence, Max-SAT instances are represented as multisets of clauses where w repetitions of clause C is equivalent to (C, w) in our notation. The main advantage of this simplification is that it can inherit from the modern SAT solver SATz [52] the efficient data structures as well as the efficient implementation of propagation procedures. This may explain why MAXSATZ seems to be better than Max-DPLL on unweighted instances. The main disadvantage of this assumption is that MAXSATZ cannot be applied to instances having mandatory clauses or clauses with high weights. As shown in Section 7, this limits severely its scope of applicability.

9. Conclusion and future work

This paper introduces a novel Max-SAT framework which highlights the relationship between SAT and Max-SAT solving techniques. Most remarkably, it extends the concept of *resolution*. Our resolution rule, first proposed in [1], has been proved complete in [27]. There are many beneficial consequences of this approach:

- It allows to talk about Max-SAT solving with the usual SAT terminology.
- It allows to naturally extend basic algorithms such as DPLL and DP.
- It allows to express several solving techniques that are spread around the Max-SAT literature with a common formalism, see their logical interpretation and see the connection with similar SAT, CSP and WCSP techniques.

From a practical point of view, we have proposed a hybrid algorithm that combines search and some restricted forms of inference. It follows a typical search strategy but, at each visited node, it attempts to simplify the current subproblem using special cases of resolution with which the problem is transformed into a simpler, equivalent one. Our experiments on a variety of domains show that our algorithm can be orders of magnitude faster than its competitors.

Our current solver lacks features that are considered very relevant in the SAT context (for example, clause learning, re-starts, etc.). Since our framework makes the connection between SAT and Max-SAT very obvious, these features should be easily incorporated in the future. Additionally, some of the ideas presented in this paper have been borrowed from the *weighted CSP* field [43]. Therefore, it seems also possible to incorporate new (weighted) constraint processing techniques.

Finally, we want to note the recent work of [33] and [51] in which very good lower bounds are obtained by temporarily setting $\top = 1$ and simulating unit propagation. Since the hyper-resolution rules presented in Section 5.2 are special cases of their more general algorithm, we want to explore if their approach can be fully described with our resolution rule.

Appendix A. Correctness and complexity of `Max-VarElim`

In this appendix we prove Lemmas 19 and 21, which establish the correctness of the `Max-VarElim` function in Fig. 5 and its time and space complexity. In the proofs we borrow some ideas from [18,21,27] and adapt them to our framework.

In the following, when we write $C \in \mathcal{F}$ we mean $(C, u) \in \mathcal{F}$ for some weight u (there is no ambiguity because all clauses in $\mathcal{F}$ are different). We use symbol $\mathcal{F} \vdash_{x_i} \mathcal{F}'$ to denote the application of a resolution step to formula $\mathcal{F}$ resulting in formula $\mathcal{F}'$, where the clashing variable was x_i. Consider the elimination of variable x_i with Function `Max-VarElim`. First of all, the formula is partitioned into two sets of clauses, $\mathcal{B}$ and $\mathcal{F}$. Then, clauses of the form $(x_i \vee A, u)$ are fetched from $\mathcal{B}$, resolved with clashing clauses until quiescence or disappearance and, finally, are discarded. Suppose that discarded clauses are stored in a set $\mathcal{D}$. Formally, we can see the execution of `Max-VarElim` as a sequence of resolution steps,

$$\mathcal{B}_o \cup \mathcal{F}_o \cup \mathcal{D}_o \vdash_{x_i} \mathcal{B}_1 \cup \mathcal{F}_1 \cup \mathcal{D}_1 \vdash_{x_i} \cdots \vdash_{x_i} \mathcal{B}_q \cup \mathcal{F}_q \cup \mathcal{D}_q,$$

where $\mathcal{D}_o = \varnothing$. For all $0 \leqslant k \leqslant q$: $\mathcal{B}_k$ is a set of clauses that contain either x_i or $\bar{x}_i$, $\mathcal{F}_k$ is a set of clauses that *do not* contain x_i neither $\bar{x}_i$, and $\mathcal{D}_k$ is a set of clauses that contain x_i. Besides, $\mathcal{B}_q$ does not have any clause with x_i. The output of `Max-VarElim` is $\mathcal{F}_q$ that, as we will prove, is essentially equivalent to the original formula. Let N_i denote the set of variables sharing clauses with x_i in the starting $\mathcal{B}$ (namely, $\mathcal{B}_o$),

$$N_i = \big\{ x_j \neq x_i \mid \exists_{C \in \mathcal{B}_o} \, x_j \in var(C) \big\}$$

and let $n_i = |N_i|$ be its cardinality. In the remaining of this appendix we will show that: the number of new clauses generated during the sequence of resolution steps is bounded by $O(3^{n_i})$ (space complexity), the number of resolution steps is bounded by $O(9^{n_i})$ (time complexity) and, from an optimal model of $\mathcal{F}_q$ we can trivially generate an optimal model of the original formula $\mathcal{B}_o \cup \mathcal{F}_o$ (correctness).

Observe that all the variables different from x_i appearing in clauses generated by the resolution process must belong to N_i because resolution does not add new variables. Therefore, all the clauses in $\mathcal{B}_k$ have the form $l \vee A$ where $var(l) = x_i$ and $var(A) \subseteq N_i$. Variable x_i must appear in the clause either as a positive or negative literal (namely, there are 2 options) and every $x_j \in N_i$ may or may not appear in A and, if it appears, it can be in positive or negative form (namely, there are 3 options). Consequently, the size of $\mathcal{B}_k$ is bounded by 2×3^{n_i}. For similar reasons, every clause C *added* to $\mathcal{F}$ during the resolution process satisfies that $var(C) \subseteq N_i$. Every $x_j \in N_i$ may or may not appear in C and, if it appears, it may be positive or negative (namely, there are 3 options). Consequently, the number of non-original clauses in $\mathcal{F}_k$ is bounded by 3^{n_i}. Therefore, the number of clauses added to $\mathcal{B}$ and $\mathcal{F}$ during the execution of `Max-VarElim` is bounded by $2 \times 3^{n_i} + 3^{n_i}$. As a result, its space complexity is $O(3^{n_i})$.

Next, we analyze the time complexity. Recall that two clauses $(x_i \vee A, u)$, $(\bar{x}_i \vee B, w) \in \mathcal{F}$ *clash* if $A \vee B$: is not a tautology (that is, $\forall_{l \in A} \bar{l} \notin B$) and, $A \vee B \in \mathcal{F}$ is not absorbed (that is, $\forall_{(C, \top) \in \mathcal{F}} C \nsubseteq A \vee B$). We say that a clause $(x_i \vee A, u)$ is *saturated* if there is no clause in $\mathcal{F}$ clashing with it. The following lemma shows that resolving on a clause, either removes the clause from the formula or reduces the number of clauses clashing with it,

Lemma 33. *Consider a resolution step* $\mathcal{P} \vdash_{x_i} \mathcal{P}'$ *where* $(x_i \vee A, u)$ *and* $(\bar{x}_i \vee B, w)$ *are the clashing clauses. Then, either* $x_i \vee A \notin \mathcal{P}'$ *or the number of clauses clashing with* $x_i \vee A$ *decreases.*

Proof. We reason by cases:

(1) If $u < w$ or $u = w < \top$ then the posterior $x_i \vee A$ has weight 0 (namely, disappears from the formula).
(2) If $u = w = \top$ then the effect of resolution is to add the resolvent to the formula ($\mathcal{P}' = \mathcal{P} \cup (A \vee B, \top)$). Then, $\bar{x}_i \vee B$ does not clash with $x_i \vee A$ anymore.
(3) If $u > w$ then $\bar{x}_i \vee B$ is replaced by $\bar{x}_i \vee B \vee \bar{A}$ in the formula. The new clause does not clash with $x_i \vee A$, because $A \vee B \vee \bar{A}$ is a tautology. $\square$

Consider the inner loop of `Max-VarElim`. It selects a clause $x_i \vee A$ and resolves it until either it disappears or it saturates. If $x_i \vee A$ saturates, it is removed from $\mathcal{B}$ and added to $\mathcal{D}$. We call this sequence of resolution steps the *processing* of $x_i \vee A$ and use symbol $\vdash^*_{x_i \vee A}$ to represent it. A consequence of the previous lemma is that the number of resolution steps required to process $x_i \vee A$ is bounded by the number of clauses clashing with it. Note that the number of clauses clashing with $(x_i \vee A, u)$ is bounded by 3^{n_i}, because clashing clauses must belong to $\mathcal{B}$ and variable x_i must occur negated. Therefore, for each iteration of the outer loop, the inner loop of `Max-VarElim` iterates at most 3^{n_i} times.

Consider now the outer loop of `Max-VarElim`. It selects a sequence of clauses $x_i \vee A_1, x_i \vee A_2, \ldots, x_i \vee A_s$ and processes them one after another. We can see this process as,

$$\mathcal{B}_o \cup \mathcal{F}_o \cup \mathcal{D}_o \vdash^*_{x_i \vee A_1} \mathcal{B}_{k_1} \cup \mathcal{F}_{k_1} \cup \mathcal{D}_{k_1} \vdash^*_{x_i \vee A_2} \cdots \vdash^*_{x_i \vee A_s} \mathcal{B}_{k_s} \cup \mathcal{F}_{k_s} \cup \mathcal{D}_{k_s}.$$

Recall that the algorithm always selects for processing a clause $x_i \vee A_j$ of minimal size (line 4). Observe that the size of the compensation clause $x_i \vee A \vee \bar{B}$ added to $\mathcal{B}$ (line 9) is larger than the clause that is being processed. As a consequence, once a clause is processed, it does not appear again in $\mathcal{B}$, which means that $\forall_{1 \leqslant j < j' \leqslant s} A_j \neq A_{j'}$. A direct consequence is that, since there are at most 3^{n_i} distinct A_j, the outer loop iterates at most 3^{n_i}. Therefore, the maximum number of iterations of the inner loop is $3^{n_i} \times 3^{n_i} = 9^{n_i}$, which means that the time complexity of the function is $O(9^{n_i})$.

Finally, we prove the correctness of `Max-VarElim`.

Lemma 34. *A saturated clause remains saturated during any sequence of resolution steps* $\vdash_{x_i}$.

Proof. Consider a resolution step $\mathcal{F} \vdash_{x_i} \mathcal{F}'$. Let $x_i \vee A$ and $\bar{x}_i \vee B$ be the clashing clauses, and let $x_i \vee C$ be a saturated clause of $\mathcal{F}$. We only need to prove that $x_i \vee C$ remains saturated in $\mathcal{F}'$. Since, $x_i \vee C$ is saturated in $\mathcal{F}$, either $C \vee B$ is a tautology or it is absorbed in $\mathcal{F}$. The only new clause in $\mathcal{F}'$ that could clash with $x_i \vee C$ is $\bar{x}_i \vee B \vee \bar{A}$. However, if $C \vee B$ was a tautology, so it is $C \vee B \vee \bar{A}$. If $C \vee B$ was absorbed in $\mathcal{F}$, so it will $C \vee B \vee \bar{A}$ in $\mathcal{F}'$. $\square$

A consequence of the previous lemma is that at the end of the sequence of resolution steps performed by `Max-VarElim` we have a formula $\mathcal{B}_{k_s} \cup \mathcal{F}_{k_s} \cup \mathcal{D}_{k_s}$ such that all its clauses are saturated.

To prove the correctness of `Max-VarElim` we only need to prove that any assignment I of $\mathcal{F}_{k_s}$ can be extended to variable x_i in a cost free-manner, taking into account the clauses $\bar{x}_i \vee B \in \mathcal{B}_{k_s}$ and the clauses $x_i \vee A \in \mathcal{D}_{k_s}$, because it means that finding the optimal assignment of $\mathcal{F}_{k_s}$ is equivalent to finding the optimal assignment of $\mathcal{B}_{k_s} \cup \mathcal{F}_{k_s} \cup \mathcal{D}_{k_s}$ which, in turn is equivalent to finding the optimal assignment of the original formula.

If $\mathcal{B}_{k_s} = \varnothing$ (resp. $\mathcal{D}_{k_s} = \varnothing$), variable x_i must be set to *true* (resp. *false*). Else, consider that there is a clause $x_i \vee A \in \mathcal{D}_{k_s}$ such that I does not satisfy A (similarly for $\bar{x}_i \vee B \in \mathcal{B}_{k_s}$). Variable x_i must be set to *true*. We show that I satisfies every $\bar{x}_i \vee B \in \mathcal{B}_{k_s}$: Clause $x_i \vee A$ is saturated, then either $A \vee B$ is a tautology or there is a clause $C \in \mathcal{F}_{k_s}$ with $C \subseteq A \cup B$. In the first case, since I does not satisfy A, and since $A \vee B$ is a tautology, this means that I satisfies B. In the second case, since I satisfies C and does not satisfy A, it must satisfy B.

Appendix B. Solving Max-SAT with pseudo-Boolean and SAT solvers

In *linear pseudo-Boolean* (LPB) problems over Boolean variables $\{x_1, \ldots, x_n\}$, values *true* and *false* are replaced by numbers 1 and 0, respectively. Literal l_i represents either x_i or its *negation* $1 - x_i$. A LPB problem is defined by a LPB objective function (to be minimized),

$$\sum_{i=1}^{n} a_i l_i, \quad \text{where } a_i \in \mathbb{Z}$$

and a set of LPB constraints,

$$\sum_{i=1}^{n} a_{ij} l_i \geqslant b_j, \quad \text{where } a_{ij}, b_j, \in \mathbb{Z}, \ x_i \in \{0, 1\}.$$

A Max-SAT formula can be encoded as a LPB problem [11] by partitioning the set of clauses into three sets: $\mathcal{H}$ contains the mandatory clauses $(C, \top)$, $\mathcal{W}$ contains the non-unary non-mandatory clauses $(C, u < \top)$ and $\mathcal{U}$ contains the unary non-mandatory clauses (l, u). For each hard clause $(C_j, \top) \in \mathcal{H}$ there is a LPB constraint $C'_j \geqslant 1$, 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 LPB constraint $C'_j + r_j \geqslant 1$, 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 is,

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

A LPB problem can be solved with a native LPB solver such as PUEBLO or with a SAT solver. In the latter case, each LPB constraint must be converted into a logic circuit. There are different possible conversions such as BDDs, adders or sorters. In our experiments we used MINISAT+ [53], a translating tool that converts each LPB constraint into the presumably more convenient circuit and solves the corresponding SAT formula with MINISAT. MINISAT+ converts the objective function of the LPB problem into another LPB constraint by setting an upper bound. The LPB problem is solved by decreasing the value of the upper bound until an infeasible SAT formula is found.

References

[1] J. Larrosa, F. Heras, Resolution in Max-SAT and its relation to local consistency for weighted CSPs, in: Proc. of the 19th IJCAI, Edinburgh, UK, 2005, pp. 193–198.
[2] J. Larrosa, F. Heras, New Inference Rules for Efficient Max-SAT Solving, in: Proc. of AAAI-06, Boston, MA, 2006.
[3] H. Xu, R. Rutenbar, K. Sakallah, sub-SAT: A formulation for relaxed boolean satisfiability with applications in routing, in: Proc. Int. Symp. on Physical Design, San Diego CA, 2002, pp. 182–187.
[4] D. Strickland, E. Barnes, J. Sokol, Optimal protein structure alignment using maximum cliques, Operations Research 53 (3) (2005) 389–402.
[5] M. Vasquez, J. Hao, A logic-constrained knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite, Journal of Computational Optimization and Applications 20 (2) (2001) 137–157.
[6] J.D. Park, Using weighted MAX-SAT engines to solve MPE, in: Proc. of AAAI-02, Edmonton, Alberta, Canada, 2002, pp. 682–687.
[7] T. Sandholm, An algorithm for optimal winner determination in combinatorial auctions, in: Proc. of the 16th IJCAI, 1999, pp. 542–547.
[8] N. Bansal, V. Raman, Upper Bounds for MaxSat: Further improved, in: ISAAC, 1999, pp. 247–258.
[9] R. Niedermeier, P. Rossmanith, New upper bounds for maximum satisfiability, J. Algorithms 36 (1) (2000) 63–88.
[10] J. Chen, I. Kanj, Improved exact algorithms for Max-SAT, in: Proc. of the 5th Latin American Symposium on Theoretical Informatics, 2002, pp. 341–355.
[11] S. de Givry, J. Larrosa, P. Meseguer, T. Schiex, Solving Max-SAT as weighted CSP, in: Proc. of CP-03, Cork, Ireland, 2003, pp. 363–376.
[12] H. Shen, H. Zhang, Study of lower bounds for Max-2-SAT, in: Proc. of AAAI-04, San Jose, CA, 2004, pp. 185–190.
[13] Z. Xing, W. Zhang, MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability, Artificial Intelligence 164 (1–2) (2005) 47–80.
[14] T. Alsinet, F. Manya, J. Planes, Improved exact solver for weighted Max-SAT, in: Proc. of SAT-05, St Andrews, Scotland, 2005, pp. 371–377.
[15] R. Nieuwenhuis, A. Oliveras, On SAT modulo theories and optimization problems, in: Proc. of SAT-06, Seattle, WA, 2006, pp. 156–169.
[16] U. Bertele, F. Brioschi, Nonserial Dynamic Programming, Academic Press, New York, 1972.
[17] R. Dechter, Constraint Processing, Morgan Kaufmann, San Francisco, CA, 2003.
[18] I. Rish, R. Dechter, Resolution vs. inference: Two approaches to SAT, Journal of Automated Reasoning 24 (1) (2000) 225–275.
[19] S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey, BIT 25 (1985) 2–23.
[20] M. Davis, G. Logemann, G. Loveland, A machine program for theorem proving, Communications of the ACM 5 (1962) 394–397.

[21] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the ACM 3 (1960).

[22] A.V. Gelder, Satisfiability testing with more reasoning and less guessing, in: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1995, pp. 0–1.

[23] F. Bacchus, Enhancing Davis Putnam with extended binary clause reasoning, in: Proc. of AAAI-02, Edmonton, Alberta, Canada, 2002, pp. 613–619.

[24] L. Drake, A. Frisch, T. Walsh, Adding resolution to the DPLL procedure for Boolean satisfiability, in: Proc. of SAT-02, Cincinnati, OH, 2002, pp. 122–129.

[25] C. Papadimitriou, Computational Complexity, Addison-Wesley, Reading, MA, 1994.

[26] T. Schiex, Arc Consistency for soft constraints, in: Proc. of CP-00, Singapore, 2000, pp. 411–424.

[27] M. Bonet, J. Levy, F. Manya, A complete calculus for Max-SAT, in: Proc. of SAT-06, Seattle, WA, 2006, pp. 240–251.

[28] B. Cha, K. Iwama, Adding new clauses for faster local search, in: Proc. of AAAI-96, Portland, OR, 1996, pp. 332–337.

[29] R. Wallace, E. Freuder, Comparative studies of constraint satisfaction and Davis–Putnam algorithms for Max-SAT problems, in: Cliques, Coloring and Satisfiability, 1996, pp. 587–615.

[30] T. Alsinet, F. Manya, J. Planes, Improved branch and bound algorithms for Max-SAT, in: Proc. of SAT-03, Portofino, Italy, 2003, pp. 408–415.

[31] C. Bessière, Arc-consistency and arc-consistency again, Artificial Intelligence 65 (1) (1994) 179–190.

[32] S. de Givry, F. Heras, J. Larrosa, M. Zytnicki, Existential arc consistency: getting closer to full arc consistency in weighted CSPs, in: Proc. of the 19th IJCAI, Edinburgh, UK, 2005, pp. 84–89.

[33] C.M. Li, F. Manya, J. Planes, Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers, in: Proc. of CP-05, Sitges, Spain, 2005, pp. 403–414.

[34] C.-M. Li, F. Manyà, J. Planes, New inference rules for Max-SAT, Journal of Artificial Intelligence Research, in press.

[35] H.M. Sheini, K.A. Sakallah, Pueblo: A hybrid pseudo-Boolean SAT solver, Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 165–189.

[36] N. Eén, N. Sörensson, An Extensible SAT-solver, in: Proc. of SAT-03, Portofino, Italy, 2003, pp. 502–518.

[37] D.S. Johnson, M. Trick, Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, AMS, 1993.

[38] J.-C. Régin, Using constraint programming to solve the maximum clique problem, in: Proc. of CP-03, Cork, Ireland, 2003, pp. 634–648.

[39] D. Wood, An algorithm for finding maximum cliques in a graph, Operations Research Letters 21 (1997) 211–217.

[40] P.R.J. Ostergard, A fast algorithm for the maximum clique problem, Discrete Applied Mathematics 120 (2002) 197–207.

[41] T. Fahle, Simple and fast: Improving a branch-and-bound algorithm for maximum clique, in: Proc. of ESA, 2002, pp. 485–498.

[42] K. Leyton-Brown, M. Pearson, Y. Shoham, Towards a universal test suite for combinatorial auction algorithms, ACM E-Commerce (2000) 66–76.

[43] P. Meseguer, F. Rossi, T. Schiex, Soft constraints, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, Amsterdam, 2006 (Ch. 9).

[44] J. Larrosa, T. Schiex, Solving weighted CSP by maintaining arc-consistency, Artificial Intelligence 159 (1–2) (2004) 1–26.

[45] R. Dechter, Bucket elimination: A unifying framework for reasoning, Artificial Intelligence 113 (1–2) (1999) 41–85.

[46] J. Larrosa, R. Dechter, Boosting search with variable elimination in constraint optimization and constraint satisfaction problems, Constraints 8 (3) (2003) 303–326.

[47] J. Larrosa, E. Morancho, D. Niso, On the practical applicability of Bucket Elimination: Still-life as a case study, Journal of Artificial Intelligence Research 23 (2005) 421–440.

[48] J. Larrosa, Node and arc consistency in weighted CSP, in: Proc. of AAAI-02, Edmonton, Alberta, Canada, 2002, pp. 48–53.

[49] M. Cooper, High-order consistency in valued constraint satisfaction, Constraints 10 (3) (2005) 283–305.

[50] E. Freuder, C. Elfe, Neighborhood inverse consistency preprocessing, in: Proc. of AAAI-96, Portland, OR, 1996, pp. 202–208.

[51] C.-M. Li, F. Manyà, J. Planes, Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT, in: Proc. of AAAI-06, Boston, MA, 2006.

[52] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proc. of the 15th IJCAI, Nagoya, Japan, 1997, pp. 366–371.

[53] N. Eén, N. Sörensson, Translating pseudo-Boolean constraints into SAT, Journal on Satisfiability, Boolean Modeling and Computation 2 (2006) 1–26.