Journal on Satisfiability, Boolean Modeling and Computation 7 (2010) 89-126
IOS Press

The Impact of Max-SAT Resolution-Based Preprocessors on Local Search Solvers

Federico Heras fheras@ucd.ie
University College Dublin
Dublin
Ireland

David Bañeres dbaneres@uoc.edu
Universitat Oberta de Catalunya
Barcelona
Spain

Abstract

In this paper we analyze three well-known preprocessors for Max-SAT. The first preprocessor is based on the so-called *variable saturation*. The second preprocessor is based on the resolution mechanism incorporated in modern branch and bound solvers. The third preprocessor is specific for the *Maximum Clique problem* and other problems with similar encoding in WCNF such as *minimum vertex covering* and *combinatorial auctions*. Our experimental investigation is divided in two parts. In the first part, we study the effect of the preprocessors on several problem instances using different metrics. In the second part, the effect of each preprocessor is analyzed in some of the most relevant Max-SAT *local search* algorithms of the literature including GSAT, WALKSAT, ADAPTNOVELTY+, IROTS and SAPS. Results indicate that some of these algorithms find much better solutions after the preprocessor. Furthermore, some preprocessed instances can be solved to *optimality* with local search under very specific conditions.

KEYWORDS: *max-SAT, stochastic local search, inference*

Submitted March 2010; revised August 2010; published November 2010

1. Introduction

The *Satisfiability* problem in propositional logic (*SAT*) is the task to decide whether a given propositional formula has a model. *Max-SAT* is the optimization variant of SAT and it can be seen as a generalisation of the SAT problem. Given a propositional formula in conjunctive normal form (*CNF*), the objective of the *unweighted Max-SAT* problem is to find a variable assignment that maximizes the number of satisfied clauses. In *weighted Max-SAT*, each clause has an associated *weight* and the goal turns into maximizing the sum of the weights of the satisfied clauses.

In this paper we focus on weighted Max-SAT which is a well-known *NP Hard* problem [32]. It is considered one of the fundamental combinatorial optimization problems and many important problems can be naturally expressed as Max-SAT. They include academic problems such as *Max-Cut* or *Max-Clique*, and real problems in domains like *routing* [46], *bioinformatics* [40], *scheduling* [45], *probabilistic reasoning* [33] and *electronic markets* [36].

Search methods for Max-SAT solving are usually classified in two categories: *systematic* and *local search*. Systematic search algorithms are exact methods that traverse the whole search space of a problem instance in a systematic way. Note that finding the optimal solution or proving that no solution exists is guaranteed. This property of systematic search is called *completeness*. On the other hand, local search is a heuristic method that initially selects a point of the search space, and moves from the current solution to a neighbour candidate. Local search solvers stop the exploration when a solution is found or as soon as some resource has been exhausted (i.e. a time limit or a maximum number of steps). Typically, local search solvers are *incomplete* and they do not guarantee to find a solution neither to prove its optimality.

An alternative way to solve problems is *inference*. The aim of inference is to make explicit some implicit information from the problem instance. Algorithms purely based on inference are complete as systematic search. However, they are usually prohibitive because of their high memory requirements [35]. In practice, limited forms of inference are applied inside search algorithms in order to reduce the search space.

Recently, a general inference mechanism for Max-SAT, denoted as resolution rule for Max-SAT, has been introduced [7, 22]. This rule which extends classical resolution [11] is *sound* and *complete*. The novel resolution rule for Max-SAT is widely used to boost systematic search algorithms [22, 18, 26].

The use of inference in local search has been restricted to the classical resolution rule and the SAT problem [9, 13, 2]. However, the benefits of the new inference mechanisms for Max-SAT are unexplored in the field of local search. In this paper, we propose to study three different resolution-based preprocessors and to analyze their effect on several local search algorithms.

The first preprocessor is based on *variable saturation* (i.e. *variable elimination*) [3]. Basically, it eliminates one-by-one a subset of variables that satisfy a certain condition. The second preprocessor is similar to the limited inference applied at each search step of a systematic search algorithm in [18, 26]. The third one is a slightly modified version of the preprocessor presented in [16] which exploits the particular structure of specific problems.

We performed an exhaustive experimental investigation. Unlike most previous works in the area, we have considered a large set of different benchmarks and local search algorithms. Our experimental investigation is divided in two parts. In the first part, we study the effect of each preprocessor on the problem instances using different metrics. In the second part, the effect of each preprocessor is analyzed in some of the most relevant local search algorithms in the literature. As a result, we have determined which preprocessors are effective in practice and which local search algorithms are more sensitive to inference.

Finally, we observed that some difficult instances in the literature such as [47] can solved to optimality with a local search algorithm after being preprocessed with a slightly modified version of the preprocessor presented in [16]. As far as we know, it is the first time that the optimal solution is found and certified for all these instances.

The structure of this paper is the following. Section 2 introduces preliminary notation and concepts about Max-SAT. Sections 3, 4 and 5 present the different preprocessors. Section 6 includes the experimental investigation. Section 7 shows how some problem instances can be solved to optimality within a SLS algorithm. The related work is discussed in Section 8. Finally, Section 9 presents some concluding remarks and points out our future work.

2. Preliminaries

In this Section we present the basic concepts needed for this paper. First, we formally introduce the Max-SAT problem and its related notation. Then, we present current complete and incomplete approaches to solve the Max-SAT problem.

2.1 Notation and Definitions

We define $X = \{x_1, x_2, \ldots, x_n\}$ as the set of Boolean variables. A *literal* is either a variable x_i or its negation $\bar{x}_i$. The variable to which a literal l refers is denoted by $var(l)$. Given a literal l, its negation $\bar{l}$ is $\bar{x}_i$ if l is x_i and it is x_i if l is $\bar{x}_i$.

A *clause* C is a disjunction of literals. Hereafter, capital letters will represent clauses. The *size* of a clause, noted $|C|$, is the number of literals that it has. The set of variables that appear in C is noted $var(C)$. A *positive* (*negative*) clause has all the literals in the positive (negative) polarity.

An *assignment* is a set of literals $A = \{l_1, l_2, \ldots, l_k\}$ such that for all $l_i \in A$, its variable $var(l_i) = x_i$ is assigned to value $true$ or $false$. 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 assigned to $false$, literal $\bar{x}_i$ is satisfied and literal x_i is falsified. If all variables in X are assigned, the assignment is called *complete*, otherwise it is called *partial*. An assignment *satisfies* a literal iff it belongs to the assignment, it satisfies a clause iff it satisfies one or more of its literals and it *falsifies* a clause iff it contains the negation of all its literals. The *empty clause* noted $\square$ has no literals (size 0) and cannot be satisfied by definition. If a clause is falsified by an assignment, the clause is *conflicting* and it can be represented using the empty clause.

A *weighted* clause is a pair (C, w), where C is a clause and w is the cost of its falsification, also called its *weight*. Many real problems contain clauses that *must* be satisfied. We call these clauses *mandatory* or *hard* and we associate to them a special weight $\top$. Note that any weight $w \geq \top$ indicates that the associated clause must be necessarily satisfied. Thus, we can replace w by $\top$ without changing the problem. Consequently, we can assume all weights in the interval $[0..\top]$. Non-mandatory clauses are also called *soft* clauses. A *weighted* formula in *conjunctive normal form* (WCNF) $\mathcal{F}$ is a set of weighted clauses.

A *model* is a complete assignment that satisfies all mandatory clauses. The *cost of an assignment* is the sum of weights of the clauses that it falsifies. Given a WCNF formula, *Weighted* Max-SAT is the problem of finding a model of minimum cost. Note that if a formula contains only mandatory clauses, weighted Max-SAT is equivalent to classical SAT. If all the clauses have weight 1, we have the (unweighted) Max-SAT problem. Hereafter, we will assume weighted Max-SAT.

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

Let u and w be two weights. Their sum is defined as,

$$u \oplus w = min\{u + w, \top\}$$

in order to keep the result within the interval $[0..\top]$.

If $u \geq w$, their subtraction is defined as,

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

Essentially, $\ominus$ is similar to the usual subtraction with the exception that $\top$ is an absorbing element.

The De Morgan rule [22] cannot be used in Max-SAT. Instead, the following rule should be repeatedly used until the conjunctive normal form is achieved:

$$(A \vee \overline{l \vee C}, w) \equiv \{(A \vee \bar{C}, w), (A \vee \bar{l} \vee C, w)\}$$

If a formula contains clauses (C, u) and (C, v), they can be replaced by $(C, u \oplus v)$ (*Aggregation*). If a formula contains the clause $(C, 0)$, such clause can be removed. The empty clause may appear in a formula. If its weight is $\top$, i.e. $(\square, \top)$, it is clear that the formula does not have any model. If its weight is w, i.e. $(\square, w)$, the cost of any assignment will include that weight, therefore w is an obvious lower bound of the formula optimal cost.

Following [22], the resolution rule can be extended from SAT to Max-SAT as,

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

where $m = \min\{u, w\}$. $(x \vee A, u)$ and $(\bar{x} \vee B, w)$ are called *clashing* clauses. $(A \vee B, m)$ is called the *resolvent*. $(x \vee A, u \ominus m)$ and $(\bar{x} \vee B, w \ominus m)$ are defined as *posterior clashing* clauses. Finally, $(x \vee A \vee \bar{B}, m)$ and $(\bar{x} \vee \bar{A} \vee B, m)$ are called *compensation* clauses.

The identification of mandatory clauses with $\top$ allows to extend some well-known simplification rules from SAT to Max-SAT such as the *subsumption* $\{(A, \top), (A \vee B, w)\} \equiv \{(A, \top)\}$ or *unit propagation*. Unit propagation for Max-SAT [22] can be applied as in SAT [11] only when unit hard clauses exist on the formula. Given a unit hard clause $(l, \top)$, literal l is *propagated* which means that it is assigned accordingly to satisfy the clause. This assignment can falsify literals in other clauses that may become also hard unit clauses. The procedure is repeated until no more hard unit clauses are generated or until a *conflict* is detected (i.e. a hard clause is falsified).

2.2 Background

In this Section we describe the existing approaches based on search and inference for Max-SAT solving.

Max-SAT solvers based on systematic search apply a natural extension of the *backtracking* algorithm [6] to handle optimization problems called *Branch and Bound* (*BB*). Two values are computed during the exploration:

- The *upper bound* (*ub*) is the sum of weights of the falsified clauses by the best complete assignment found so far.

- The *lower bound* (*lb*) is the sum of all the falsified clauses by the current partial assignment plus an underestimation of the weight of the clauses that will become unsatisfied by extending the current partial assignment.

The *lb* and *ub* significantly reduce the search space by pruning useless regions when $lb \geq ub$. Modern Max-SAT solvers [26, 22, 18, 28] improve the lower bound by applying a limited number of Max-SAT resolution steps during the search.

Regarding incomplete methods, many *Stochastic Local Search* (*SLS*) methods have been proposed for SAT. The main drawback of SLS algorithms is that they usually fall in *local minima*, that is, areas of the search space where no improvements can be reached.

Gsat [38] was one of the first local search methods for SAT. The algorithm starts by assigning a random Boolean value to each variable. If the assignment satisfies all clauses, the algorithm terminates, returning the assignment. Otherwise, GSAT *flips* the variable (i.e. the value of such variable is changed from *true* to $false$ or vice versa) that minimizes the total number of unsatisfied clauses. Later, major improvements were obtained with the development of the Walksat architecture [37] and its variants. Basically, the difference appears on the selection of the variable to flip. At each search step, the Walksat algorithm chooses a currently unsatisfied clause and then *flips* a variable occurring in this clause. Further research on the Walksat architecture resulted in the introduction of sophisticated schemes for selecting the variable to be flipped.

Other methods based on *Stochastic Local Search* (*SLS*) have been also proposed for SAT. They have been extended to unweighted Max-SAT by keeping track of the best solution found so far in the search process. Similar methods have been developed directly for unweighted and, in particular, weighted Max-SAT. Current SLS algorithms include other techniques based on *Dynamic Local Search* (DLS), *tabu search* and *iterated local search*. Dynamic Local Search is based on the idea of modifying an evaluation function in order to prevent the search from getting stuck in local minima [43]. Tabu search stores in a (tabu) list the most recent assignments in order to avoid revisiting them. Good performance was reported for Reactive Tabu Search (H-RTS), a tabu search that dynamically adjusts the tabu list size on unweighted Max-SAT instances [5]. Iterated Local Search (*ILS*) consists in alternating between local search and *perturbation* phases which are designed to take the search away from the local minima reached by the subsidiary local search procedure. For instance, ILS [50] uses a local search algorithm based on 2 and 3-flip neighbourhoods.

The use of inference in local search is restricted to the classical resolution rule and the SAT problem. In [9] a restricted form of resolution procedure is presented, which adds new clauses based on unsatisfied clashing clauses at the local minima. More recently in [13], authors implemented the same idea along with other techniques in a complete local search solver. Finally, authors in [2] improved the performance of modern SLS solvers for SAT with a resolution-based preprocessor.

In this paper, we will study the effect of existing preprocessors on different SLS algorithms that include all the described techniques. In particular, we considered the following algorithms: Gsat [38] , Walksat [37], Adaptnovelty+ [19] (a Walksat variant), Irots [39] (it applies tabu search and iterated local search) and Saps [43] (a dynamic local search algorithm).

Algorithm 1: Max-DP Algorithm

```
Function VarElim($\mathcal{F}, \top, x_i$) : WCNF formula
1    $\mathcal{B} := \{(C,u) \in \mathcal{F} |\ x_i \in var(C)\}$;
2    $\mathcal{F} := \mathcal{F} - \mathcal{B}$;
3    while $\exists(x_i \vee A, u) \in \mathcal{B}$ do
4        $(x_i \vee A, u)$ :=GetClause($\mathcal{B}$);
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   return $\mathcal{F}$ ;
Function Max-DP($\mathcal{F}, \top$) : $\mathbb{N}$
12   $\mathcal{F}$ := Simplify($\mathcal{F}, \top$);
13   if $\mathcal{F} = \emptyset$ then return 0;
14   if $\mathcal{F} = \{(\Box, u)\}$ then return $u$;
15   $x_i$ :=SelectVar($\mathcal{F}$);
16   return Max-DP(VarElim($\mathcal{F}, \top, x_i$),$\top$);
```

Algorithm 2: Variable Saturation Preprocessor

```
Function Preprocessor($\mathcal{F}, \top, K$) : $\mathbb{N}$ or $\mathcal{F}$
1    $\mathcal{F}$ := Simplify($\mathcal{F}, \top$);
2    if $\mathcal{F} = \emptyset$ then return 0 ;
3    if $\mathcal{F} = \{(\Box, u)\}$ then return $u$ ;
4    $x_i$ :=SelectVar($\mathcal{F}, K$) ;
5    $d_i$ :=Degree($\mathcal{F}$,$x_i$) ;
6    if $d_i <= K$ then return Preprocessor(VarElim($\mathcal{F}, \top, x_i$),$\top$,$K$);
7    return $\mathcal{F}$ ;
```

3. Variable Saturation Preprocessor

The preprocessor introduced in this Section is similar to the one in [3]. It is based on Max-DP [7, 22] which is a complete inference algorithm for Max-SAT. Max-DP is an extension of the Davis and Putnam algorithm [11] for Max-SAT. It is illustrated in Algorithm 1. The main difference between them is that Max-DP performs the resolution rule for Max-SAT instead of the classical resolution. At each recursive call Max-DP applies simplification rules such as *subsumption*, *aggregation* and *unit propagation* (line 12).

Then, variables are *selected* according to some heuristic and *eliminated* one-by-one (lines 15 - 16) until the empty formula or an empty clause is obtained (lines 13 - 14).

Function *VarElim* performs the *elimination* (or *saturation*) of a variable x_i from the formula $\mathcal{F}$. First, function *VarElim* computes the *bucket* of x_i, noted $\mathcal{B}$, which contains the set of clauses containing variable x_i. Then, each clause $(x_i \vee A, u) \in \mathcal{B}$ is selected (line 4) and it is resolved iteratively with all its clashing clauses (line 5). Resolvents are added to the weighted formula $\mathcal{F}$ while compensation clauses are added to the current bucket $\mathcal{B}$ (lines 6 - 10). This process stops when the weight u of the clause $(x_i \vee A, u)$ decreases to 0 or no clashing clauses exist.

Solving a Max-SAT instance by successively eliminating all the variables is not competitive as we will see in the experimental results. In [3] the number of variables to be eliminated is restricted given a parameter K and considering the *constraint graph* of the problem instance and the *degree* of each a variable.

Let us define *constraint graph* and *degree* of a variable to further explain the variable saturation preprocessor. In a *constraint graph*, the nodes represent the Boolean variables occurring in the problem instance, and an edge is added between two vertices if the variables of the vertices occur in the same clause. The *degree* of a variable x_i is the number of adjacent nodes of the variable x_i in its constraint graph.

The *Variable Saturation Preprocessor* [3] is shown in Algorithm 2. The algorithm is similar to Max-DP. The main differences are:

- The preprocessor returns two possible results: a natural number $\mathbb{N}$ if all variables have been successfully eliminated or a formula $\mathcal{F}$ if only a subset of variables have been eliminated.
- The preprocessor selects a variable x_i with degree smaller or equal to K (lines 4-5).
- The degree of the selected variable x_i limit the number of steps of the preprocessor. The idea is to saturate variables in which the application of variable saturation is not very costly in terms of time and space. Therefore, if the degree of variable x_i is smaller or equal to K, the variable x_i is eliminated (line 6), otherwise the current formula is returned (line 7) and the algorithm stops.

4. Unit Propagation Preprocessor

The objective of the *Unit Propagation* (UP) Preprocessor is to derive new empty clauses $(\square, w)$ through a resolution process similar to the one proposed in [18] (See Algorithm 3).

Initially, `UP-Preprocessor` replaces each occurrence of (l, u) and $(\bar{l}, w)$ by $(l, u \ominus m)$, $(\bar{l}, w \ominus m), (\square, m)$ with $m = \min\{u, w\}$ (line 9).

Then, `UP-Preprocessor` executes a *simulation of unit propagation* (labelled as `SUP` in lines 10 and 15) to find a conflicting clause. The simulation considers the soft clauses as hard and, consequently, their weights are omitted during the resolution (line 1). The `SUP` procedure uses a modified FIFO (*First In First Out*) queue Q to store all unit clauses (line 2). Q is a non-standard queue to handle literals pending of propagation. Unlike classical queues where after fetching an element, it is removed, here the element is only marked as explored. Hence, Q contains the already propagated literals and the pending ones. Moreover, each literal l in Q is associated with the original clause that caused its propagation and it is called the *reason* of l.

While non-propagated literals exist in Q (line 3), the oldest non-propagated literal l is extracted from Q and it is marked as propagated (line 4). Then, all clauses containing $\bar{l}$ are

```
Algorithm 3: Unit Propagation Preprocessor
   Function SUP(F) : Queue
1      Assume all clauses in $\mathcal{F}$ are hard ;
2      InitQueue($Q$) ;
3      while ($Q$ contains non-propagated literals) do
4          $l :=$ GetFirstNonPropagatedLiteral($Q$) ;
5          foreach clause $C \vee \bar{l}$ that becomes unit or falsified do
6              if $C \vee \bar{l}$ becomes a unit $q$ then Enqueue($Q, q$) ;
7              else if $C \vee \bar{l}$ becomes falsified then return $Q$ ;
8      return $\emptyset$ ;
   Procedure UP-Preprocessor()
9      Replace $\{(l, v), (\bar{l}, w)\} \in \mathcal{F}$ by $\{(l, v \ominus m), (\bar{l}, w \ominus m), (\Box, m)\}$ and
       $m := \min(v, w)$ ;
10     $Q :=SUP(\mathcal{F})$ ;
11     while $Q \neq \emptyset$ do
12         $\Upsilon :=$ BuildTree(Q) ;
13         $m :=$ minimum weight among clauses in $\Upsilon$;
14         ApplyResolution( $\Upsilon$, $m$ ) ;
15         $Q :=SUP(\mathcal{F})$ ;
```

traversed checking if they become falsified or unit (line 5) . If one of the clauses becomes a unit clause q, it is enqueued in Q to be propagated later (line 6). The procedure iterates until:

- A conflicting clause is found (line 7) and the current state of the Q queue is returned.
- There are no more literals to propagate and an empty queue is returned (line 8) which means no conflict was found.

If SUP yields a conflict (line 7), it means that there is a subset $\mathcal{F}'$ of clauses that cannot be simultaneously satisfied. $\mathcal{F}'$ can be determined using the information provided by Q. Since $\mathcal{F}'$ is unsatisfiable, the empty clause $\Box$ can be derived from $\mathcal{F}'$ via resolution. Such resolution process is called a *refutation*. A refutation for an unsatisfiable clause set $\mathcal{F}'$ is a *resolution refutation tree* (or simply a *refutation tree*) if every clause is used exactly once during the resolution process.

The BuildTree process builds the refutation tree from the propagation queue Q (line 12) as follows: let C_0 be the conflicting clause. *Reasons* associated to literals in Q are traversed in a *LIFO* (*Last In First Out*) fashion until a clashing clause D_0 is found. Then resolution is applied between C_0 and D_0, obtaining resolvent C_1. Next, the traversal of Q continues until a clause D_1 that clashes with C_1 is found, giving resolvent C_2 and the process iterates until the obtained resolvent is the empty clause $\Box$.

If we take into account again the weights of the clauses and actually apply Max-SAT resolution as dictated by Υ, it will produce a new clause $(\Box, m)$, where m is the minimum weight among all the clauses in the tree Υ (line 13). It is important to remark that at each step in the Max-SAT resolution process we do not consider the minimum weight of each pair of clashing clauses but rather the minimum of all the clauses in the resolution tree. This is why m is passed as a parameter in line 14.

Before the `UP-Preprocessor`, *Probing* [25, 18] is also applied in order to generate unit clauses. The idea is to temporarily assume that l is a unit clause and then *simulate* unit propagation (*i.e.*, execute `SUP()`). If unit propagation does not derive a conflict, no action is carried out. Otherwise, we build the resolution tree Υ from the propagation queue Q. If all the clauses in Υ are hard, we know that $\bar{l}$ must be added to the assignment. Otherwise, we can reproduce Υ applying Max-SAT resolution with the weighted clauses and derive a unit clause $(\bar{l}, m)$ where m is the minimum weight among the clauses in Υ. Having unit soft clauses upfront makes the future executions of `UP-Preprocessor` much more effective in the subsequent search. Besides, if we derive both (l, u) and $(\bar{l}, w)$ we can generate via resolution a new empty clause $(\Box, min\{u, w\})$.

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

Observe that subscripts α, β, γ, δ, ϵ and φ are used to identify clauses. We note $l(\alpha)$ as the reason α associated to literal l. Initially, queue Q is empty $Q = [\|]$. Hereafter, symbol $\|$ inside Q separates propagated literals from non-propagated literals. The steps of the `UP-Preprocessor` onto the formula $\mathcal{F}$ are described next:

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

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

- Apply Max-SAT resolution. *We apply Max-SAT resolution as indicated by the refutation tree computed in the previous step. The resulting formula $\mathcal{F}' = \{(x_1 \vee x_2, \top), (x_1 \vee \bar{x}_5, 1), (\Box, 2), (x_1 \vee \bar{x}_2 \vee \bar{x}_3, 1), (x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4, 2), (x_1 \vee x_2 \vee x_3 \vee \bar{x}_4, 2)\}$.*

5. Clique Preprocessor

In this Section we present a preprocessor which can be applied on problems with a very specific structure. We refer to these problems as *binary unate covering problems* (*BUCP*). First, we introduce the BCUP which is mainly composed by a set of hard binary clauses and a set of soft unit clauses. Next, two resolution-based rules, the *Star* and the *Unit Rule*, are presented. Finally, the preprocessor particularly suitable for binary unate covering problems is described. We will refer to it as *Clique Preprocessor* (*Cl*) because the initial application was intended for the Maximum Clique problem [16].

5.1 Binate, Unate and Binary Unate Covering Problem

Following [34, 10], consider a problem instance P of the form

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

subject to $A \cdot x \geq b, \quad x \in \{0,1\}^n$

where c_j is a non-negative integer cost associated with variable x_j, $1 \leq j \leq n$ and $A \cdot x \geq b, x \in \{0,1\}^n$ denote the set of m linear constraints. If every entry a_{ij} with $1 \leq i \leq m$ and $1 \leq j \leq n$ in the (m x n) matrix A is in the set $\{-1,0,1\}$ and $b_i = 1, 1 \leq i \leq m$, then P is an instance of the *binate covering problem* (BCP). Differently, if every entry a_{ij} in the matrix A is in the set $\{0,1\}$, then P is an instance of the *unate covering problem* (UCP). Finally, if P is a unate covering problem and $\forall_{i=1}^{m} \sum_{j=1}^{n} a_{ij} = 2$, that is, each linear constraint involves exactly two different variables, the problem P is an instance of the *binary unate covering problem* ($BUCP$).

The binary unate covering problem can be reformulated as a weighted Max-SAT problem.

- For each linear constraint $i, 1 \leq i \leq m$ that involves variables x_i and y_i, a new binary hard clause $(x_i \vee y_i, \top)$ is added.
- The objective function is transformed into a set of weighted soft clauses. Each term $c_j \cdot x_j$ becomes a new soft unit clause $(\bar{x}_j, c_j)$.

In other words, the Max-SAT reformulation of the binary unate covering problem consists in a set of binary hard clauses and a set of soft unit clauses. Observe that prominent optimization problems can be modelled as instances of the binary unate covering problem such as the *Maximum Clique*, *Maximum Independent set*, *Minimum Vertex Covering* and *Combinatorial Auctions*, among others.

5.2 Star Rule

The Star Rule [1, 22, 16] can be used to create new empty clauses from a long clause and a set of appropriate unit clauses.

$$\{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik}, w_0), (x_{i1}, w_1), (x_{i2}, w_2), ..., (x_{ik}, w_k)\} \equiv$$

$$\left\{ \begin{array}{l} (\Box, m), (\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik}, w_0 \ominus m), \\ (x_{i1}, w_1 \ominus m), (x_{i2}, w_2 \ominus m), \ldots, (x_{ik}, w_k \ominus m), \\ (x_{i1} \vee \overline{\bar{x}_{i2} \vee \bar{x}_{i3} \vee \cdots \vee \bar{x}_{ik}}, m), \\ (x_{i2} \vee \overline{\bar{x}_{i3} \vee \bar{x}_{i4} \vee \cdots \vee \bar{x}_{ik}}, m), \\ (x_{i3} \vee \overline{\bar{x}_{i4} \vee \bar{x}_{i5} \vee \cdots \vee \bar{x}_{ik}}, m), \\ \ldots, \\ (x_{ik-1} \vee x_{ik}, m) \end{array} \right\}$$

where $m = min\{w_0, w_1, ..., w_k\}$.

Note that the Star Rule generates a new empty clause and it is extremely effective when a large number of unit clauses are available.

Example 2 *Consider the initial formula* $\{(\bar{x}_1 \vee \bar{x}_2, 1), (x_1, 1), (x_2, 1)\}$. *This example shows each step of resolution needed to obtain the same result provided by the Star Rule. First, the resolution rule is applied between the first and the third clauses and the formula* $\{(x_1 \vee x_2, 1), (x_1, 1), (\bar{x}_1, 1)\}$ *is obtained. Finally, the resolution rule between the second and the third clauses produces* $\{(x_1 \vee x_2, 1), (\Box, 1)\}$.

5.3 Unit Rule

The Unit Rule [16] can be used to create new unit clauses from a long clause and a set of appropriate binary hard clauses. Given a subset of variables $\{x_{i1}, x_{i2}, \ldots, x_{ik}, x_j\} \subseteq X$, consider the following subset of binary hard clauses:

$$Bin(x_{i1}, x_{i2}, ..., x_{ik}, x_j) \equiv \{(x_{i1} \vee x_j, \top), (x_{i2} \vee x_j, \top), ..., (x_{ik} \vee x_j, \top)\}$$

The Unit Rule has the form:

$$\{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik}, w), Bin(x_{i1}, x_{i2}, ..., x_{ik}, x_j)\} \equiv$$
$$\{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik} \vee \bar{x}_j, w), Bin(x_{i1}, x_{i2}, ..., x_{ik}, x_j), (x_j, w)\}$$

Example 3 *Consider the initial formula* $\{(\bar{x}_1 \vee \bar{x}_2, 1), (x_1 \vee x_3, \top), (x_2 \vee x_3, \top)\}$. *This example shows each step of the resolution needed to obtain the same result provided by the Unit Rule. First, the resolution rule between the first and the second clauses is applied and the formula* $\{(\bar{x}_2 \vee x_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, 1), (x_1 \vee x_3, \top), (x_2 \vee x_3, \top)\}$ *is obtained. The application of the resolution rule between the first and the last clauses produces* $\{(x_3, 1), (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3, 1), (x_1 \vee x_3, \top), (x_2 \vee x_3, \top)\}$.

5.4 The Preprocessor

In this Section, we present the Clique Preprocessor that exploits the synergy between the Unit and the Star Rule. The Unit Rule generates unit positive clauses from negative clauses and binary positive hard clauses. These unit clauses are used by the Star Rule which transforms them into empty clauses. The preprocessor works in an on-demand fashion: it triggers the Unit Rule only if it is guaranteed the subsequent execution of the Star Rule. Before introducing the details of the preprocessor, we present a useful definition.

Definition 1 *A negative clause* $(C, w) = (\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik}, w)$ *is unit-related with respect to literal* x', *denoted as* $(C, w)_{x'}$, *if and only if* $Bin(x_{i1}, x_{i2}, \ldots, x_{ik}, x') \in \mathcal{F}$.

Observe that we can always apply the Unit Rule to a clause C unit-related with respect to a literal x in order to generate a new positive unit soft clause (x, w). The basic idea of the preprocessor is to generate the appropriate unit clauses with the Unit Rule so that we can apply the Star Rule later in order to produce an empty clause. Note that, the final objective is to increase as much as possible the weight of the empty clause.

The preprocessor is shown in Algorithm 4. It iterates over all the negative clauses (line 2). For each negative clause (C, w_0) the algorithm searches for one unit clause for each literal in C. To do so, for each literal l_i in C the algorithm seeks a clause (C', w_i) unit-related with respect to l_i (i.e. $(C', w_i)_{l_i}$) and stores it in the structure S. Note that

Algorithm 4: Clique Preprocessor

```
Function Clique-Preprocessor($\mathcal{F}$)
1   while ¬ Quiescence do
2       foreach negative clause $(C, w_0) = (\bar{l}_1 \vee \bar{l}_2 \vee ... \vee \bar{l}_k, w_0) \in \mathcal{F}$ do
3           $S := \emptyset$ ;
4           foreach $l_i \in C$ do
5               if $\exists (C', w_i) \in \mathcal{F}$ s.t. $(C', w_i) \neq (C, w_0) \wedge (C', w_i) \notin S \wedge (C', w_i)_{l_i}$ then
6                   $S := S \cup ((C', w_i), l_i)$ ;
7           if $|S| = k$ then
8               foreach $((C', w_i), l_i) \in S$ s.t. $(C', w_i) = (\bar{l'}_1 \vee \bar{l'}_2 \vee ... \vee \bar{l'}_p, w_i)$ do
9                   Apply Unit Rule to $(C', w_i), Bin(\bar{l'}_1, \bar{l'}_2, ..., \bar{l'}_p, l_i)$ ;
10              Apply Star Rule to $\{(l_1, w_1), (l_2, w_2), ..., (l_k, w_k), (C, w_0)\}$ ;
11              Insert new resolvents on $\mathcal{F}$ ;
```

all the negative clauses inserted in S must be different, and they must be also different from the initial (C, w_0) (lines 3-6). If a unit-related clause is found for each literal in C (line 7), the algorithm applies the two simplification rules. First, for each pair in structure S, it applies the Unit Rule in order to create the necessary unit clauses (lines 8-9). Once all unit clauses have been generated, the algorithm proceeds to apply the Star Rule (line 10).

The presented preprocessor is similar to the algorithm presented in [16] with two slight differences. The first one is that the algorithm is applied until quiescence (line 1). In other words, the algorithm is repeated until there is no more increments on the empty clause. The second one is that the algorithm shows explicitly that new resolvents are added to the formula (line 11). This point will become relevant in the experimental investigation (See Section 6.4).

Recall that this preprocessor is focused to problems that contain basically negative soft units and positive hard binary clauses in the original formula (the other clauses are ignored by the algorithm). Therefore, one can easily see that, at any point of the execution of Algorithm 4, each negative clause (C, w) is in $\mathcal{F}$ because either (i) (C, w) is an initial unit soft clause in $\mathcal{F}$ or (ii) (C, w) was generated by some application of the Unit Rule. This observation leads to the following lemma.

Lemma 1 *Within the Clique preprocessor, all the compensation clauses in $\mathcal{F}$ generated by the Star Rule are subsumed by binary hard clauses in $\mathcal{F}$.*

Proof 1 *Suppose that the Star Rule is applied to an arbitrary subset of clauses in $\mathcal{F}$:*

$$\{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee \dots \bar{x}_{ik}, m), (x_{i1}, m), (x_{i2}, m), \dots (x_{ik}, m)\}$$

The resulting empty clause and resolvent clauses are $\{(\square, m), (\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik}, m)\}$ while the compensation clauses are

$$\left\{\begin{array}{l}(x_{i1} \vee \overline{\bar{x}_{i2} \vee \bar{x}_{i3} \vee \cdots \vee \bar{x}_{ik}}, m),\\ (x_{i2} \vee \overline{\bar{x}_{i3} \vee \bar{x}_{i4} \vee \cdots \vee \bar{x}_{ik}}, m),\\ (x_{i3} \vee \overline{\bar{x}_{i4} \vee \bar{x}_{i5} \vee \cdots \vee \bar{x}_{ik}}, m),\\ \ldots,\\ (x_{ik-1} \vee x_{ik}, m)\end{array}\right\}$$

Observe that $(\bar{x}_{i1} \vee \bar{x}_{i2} \vee \ldots \bar{x}_{ik}, m) \in \mathcal{F}$ because the following set of Unit Rules were applied (in reverse order):

$$\begin{array}{rcl}\{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee \ldots \vee \bar{x}_{ik-1}, m) & , & Bin(x_{i1}, x_{i2}, \ldots, x_{ik-1}, x_{ik})\}\\ \{(\bar{x}_{i1} \vee \bar{x}_{i2} \vee ... \vee \bar{x}_{ik-2}, m) & , & Bin(x_{i1}, x_{i2}, \ldots, x_{ik-2}, x_{ik-1})\}\\ & \cdots & \\ \{(\bar{x}_{i1} \vee \bar{x}_{i2}, m) & , & Bin(x_{i1}, x_{i2}, x_{i3})\}\\ \{(\bar{x}_{i1}, m) & , & Bin(x_{i1}, x_{i2})\}\end{array}$$

Moreover, note that:

- *The set $Bin(x_{i1}, x_{i2})$ subsumes all compensation clauses $(x_{i1} \vee \overline{\bar{x}_{i2} \vee \bar{x}_{i3} \vee \cdots \vee \bar{x}_{ik}}, m)$.*
- *The set $Bin(x_{i1}, x_{i2}, x_{i3})$ subsumes all compensation clauses $(x_{i2} \vee \overline{\bar{x}_{i3} \vee \cdots \vee \bar{x}_{ik}}, m)$.*
- *...*
- *The set $Bin(x_{i1}, x_{i2}, \ldots, x_{ik-1}, x_{ik})$ subsumes the compensation clause $(x_{ik-1} \vee x_{ik}, m)$.*

Hence, we can conclude that the sets of binary clauses used at each application of the Unit Rule are enough to subsume all the compensation clauses produced by the initial Star Rule.

Next, a small example of both rules working together is presented.

Example 4 *Consider the formula $\mathcal{F} = \{(\bar{x}_1, 1), (\bar{x}_2, 1), (\bar{x}_3, 1), (\bar{x}_4, 1), (\bar{x}_5, 1), (\bar{x}_6, 1), (x_1 \vee x_2, \top), (x_1 \vee x_3, \top), (x_1 \vee x_4, \top), (x_2 \vee x_5, \top), (x_2 \vee x_6, \top), (x_3 \vee x_4, \top), (x_5 \vee x_6, \top)\}$.*
A possible execution of the algorithm applies the following transformations:

- *The Unit Rule is applied to $\{(\bar{x}_1, 1), (x_1 \vee x_2, \top)\}$ which are replaced by $\{(x_1, 1), (\bar{x}_1 \vee \bar{x}_2, 1), (x_1 \vee x_2, \top)\}$. Then, the Star Rule is applied to $\{(x_1, 1), (\bar{x}_1, 1)\}$ and they are replaced by $(\square, 1)$.*
- *The Unit and Star Rules are applied to $\{(\bar{x}_3, 1), (\bar{x}_4, 1), (x_3 \vee x_4, \top)\}$ and they are replaced by $\{(\bar{x}_3 \vee \bar{x}_4, 1), (x_3 \vee x_4, \top), (\square, 2)\}$.*
- *The Unit and Star Rules are applied to $\{(\bar{x}_5, 1), (\bar{x}_6, 1), (x_5 \vee x_6, \top)\}$ and they are replaced by $\{(\bar{x}_5 \vee \bar{x}_6, 1), (x_5 \vee x_6, \top), (\square, 3)\}$.*

The current formula is $\mathcal{F} = \{(\bar{x}_1 \vee \bar{x}_2, 1), (\bar{x}_3 \vee \bar{x}_4, 1), (\bar{x}_5 \vee \bar{x}_6, 1), (x_1 \vee x_2, \top), (x_1 \vee x_3, \top), (x_1 \vee x_4, \top), (x_2 \vee x_5, \top), (x_2 \vee x_6, \top), (x_3 \vee x_4, \top), (x_5 \vee x_6, \top), (\square, 3)\}$. Observe that the unit propagation preprocessor is able to obtain a similar empty clause and formula until this point but it cannot derive more empty clauses because no unit clauses are available.
The Clique preprocessor continues:

- *The Unit Rule is applied to* $\{(\bar{x}_3 \vee \bar{x}_4, 1), (x_1 \vee x_3, \top), (x_1 \vee x_4, \top)\}$ *and they are replaced by* $\{(\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4, 1), (x_1 \vee x_3, \top), (x_1 \vee x_4, \top), (x_1, 1)\}$.

- *The Unit Rule is applied to* $(\bar{x}_5 \vee \bar{x}_6, 1), (x_2 \vee x_5, \top), (x_2 \vee x_6, \top)$ *and they are replaced by* $\{(\bar{x}_2 \vee \bar{x}_5 \vee \bar{x}_6, 1), (x_2 \vee x_5, \top), (x_2 \vee x_6, \top), (x_2, 1)\}$.

- *The Star Rule is applied to* $\{(\bar{x}_1 \vee \bar{x}_2, 1), (x_1 \vee x_2, \top), (x_1, 1), (x_2, 1)\}$ *and they are replaced by* $\{(x_1 \vee x_2, \top), (\square, 4)\}$.

The resulting formula is $\mathcal{F} = \{(\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4, 1), (\bar{x}_2 \vee \bar{x}_5 \vee \bar{x}_6, 1), (x_1 \vee x_2, \top), (x_1 \vee x_3, \top), (x_1 \vee x_4, \top), (x_2 \vee x_5, \top), (x_2 \vee x_6, \top), (x_3 \vee x_4, \top), (x_5 \vee x_6, \top), (\square, 4)\}$.

The above paragraphs and examples are focused on the description of the algorithm associated to the *UP* and *Clique* preprocessors. Hereafter, we highlight their effect on the resulting formula for the binary unate covering problem. In particular, we compare the original and the resulting formula after executing Unit Propagation and Clique preprocessors.

Remark 1 *The original encoding of the binary unate covering problem contains a large set of binary hard clauses* $(x_i \vee y_i, \top)$ *and a set of soft unit clauses* $(\bar{x}_j, w_j)$.

When the *UP* preprocessor is applied to the binary unate covering problem, it is able to detect several small inconsistent subsets involving two unit soft clauses (i.e. $(\bar{x}_i, w)$ and $(\bar{y}_i, v)$) and a hard clause $(x_i \vee y_i, \top)$. As a result, a new empty clause is produced and new binary soft clauses are added.

Remark 2 *The resulting formula after the UP preprocessor contains a large set of binary hard clauses (i.e.* $(x_i \vee y_i, \top)$*), a set of binary soft clauses (i.e.* $(\bar{x}_i \vee \bar{y}_i, w)$*) and an empty clause. If the weight of the original unit soft clauses are not consumed during the preprocessor, it may also contain original soft unit clauses with a smaller weight. UP preprocessor may generate larger soft clauses but it is unusual.*

The main difference of the *UP* preprocessor and the *Clique* preprocessor is that the later is able to detect inconsistencies involving larger soft clauses rather than only unit soft clauses. This is the main reason why the *Clique* preprocessor obtains empty clauses with higher weights as we will see in the experimental investigation.

Remark 3 *The resulting formula after the Clique preprocessor contains a large set of binary hard clauses (i.e.* $(x_i \vee y_i, \top)$*), a set of large soft clauses (i.e.* $(\bar{x}_{i1} \vee \bar{x}_{i2} \vee \bar{x}_{i3} \vee \cdots \vee \bar{x}_{in}, w)$*) and an empty clause. If the weight of the original soft unit clauses are not consumed during the preprocessor, it may also contain original soft unit clauses with a smaller weight.*

The interested reader can find more examples of the resulting formula after the Clique Preprocessor is applied in the instances contained in folder *PROTEIN_INS* of the *Partial Max-SAT Industrial Track* benchmarks used in the 2009 and 2010 Max-SAT Evaluations[1].

1. http://www.maxsat.udl.cat/

6. Empirical results

In this Section, we present the empirical evaluation of the different preprocessors. First, we describe the problem instances that were considered in our experiments. Next, we show the effect of each preprocessor in those instances. Finally, we study the performance of local search solvers with the new preprocessed instances. Preprocessors were coded in C++. All experiments were conducted on a 3Ghz Intel Pentium computer with 1GB of memory and Linux (unless differently indicated).

6.1 Benchmarks

We have considered a large set of instances of different benchmarks. These benchmarks have been used in several previous works and they have been also included on previous Max-SAT Evaluations [4, 17]. This selection will provide a large evidence of the performance of the preprocessors on problems with very different structure.

The considered problem instances can be classified as *unweighted Max-SAT* including *Max-2-SAT* and *Max-Cut*, *binary unate covering problem* reformulated as weighted Max-SAT including the *Maximum Clique Problem* and the *Combinatorial Auction Problem* and *Binate Covering Problem* including the *Max-One Problem*.

6.1.1 The Random Max-K-SAT Problem

Given a CNF formula, *Max-K-SAT* is the problem of finding a complete assignment with a maximum number of satisfied clauses. K indicates the length of all the clauses. In our experiments, we used Max-2-SAT instances with a fixed number of variables (100) and varying the number of clauses (from 300 to 1000, step 100). Note that, if we formulate the problem as weighted Max-SAT, all clauses have weight equal to 1. We generated 10 instances for each parameter combination. We used the free generator *Cnfgen* [44]. Note that this generator prevents duplication or insertion of complementary literals in clauses but not duplication of clauses.

6.1.2 The Max-Cut problem

Let be $G = (V, E)$ a *graph* where V is the set of *vertices* and E is the set of *edges* defined between pairs of vertices. 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 $v_j \in V - U$. *Max-Cut*, which is a NP-Hard problem, consists in finding a cut of maximum size. The problem can be easily modelled as unweighted Max-SAT or weighted Max-SAT with all clauses with weight equal to 1. One variable x_i is associated to each graph vertex v_i. Value $true$ ($false$) indicates that vertex v_i belongs to U (to $V - U$). For each edge (v_i, v_j), there are two clauses $(x_i \vee x_j, 1)$ and $(\bar{x}_i \vee \bar{x}_j, 1)$. Given a complete assignment, the number of violated clauses is $|E| - S$ where S is the size of the cut associated to the assignment. As a consequence, the optimal Max-SAT assignment represents the optimal Max-Cut. In our experiments, we have generated Max-Cut instances from random graphs with a fixed number of nodes (60) and varying the number of edges (from 200 to 500 step 100). 10 instances were created for each parameter combination.

6.1.3 The Minimum Vertex Covering Problem

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 (Min-Vertex-Covering) problem consists in finding a covering of minimal size.

The minimum vertex covering problem is a well-known NP-Hard problem that can be naturally formulated as weighted Max-SAT. One variable x_i is associated to each graph vertex. Value *true* ($false$) indicates that vertex x_i belongs to U (to $V - U$). There is a binary weighted clause $(x_i \vee x_j, \top)$ for each edge $(v_i, v_j) \in E$. 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.

6.1.4 The Maximum Clique Problem

Given a graph $G = (V, E)$, a clique is a set $U \subseteq V$ such that for every vertex $v \in U$, v is connected to all the vertices in U. The size of a clique is $|U|$. The maximum clique problem (Max-Clique) consists in finding a clique of maximal size.

The maximum clique problem is a well-known NP-Hard problem. As noted in [12], finding the maximum clique of a graph $G = (V, E)$ is equivalent to find a minimum vertex covering of the *complementary* graph $\bar{G}$. Given a graph $G = (V, E)$, its complementary graph is denoted by $\bar{G} = (V, \bar{E})$. It is constructed with the same set of vertices V and $(v_i, v_j) \in \bar{E}$ iff $(v_i, v_j) \notin E$. Hence, we can model Max-Clique problems as Minimum Vertex Covering problems over the complementary graph. Observe that the maximum size of the maximum clique is equivalent to $|V| - S$, where S is the size of the minimum vertex covering.

In our experiments, we considered random generated Max-Clique instances with 150 variables and varying the number of edges from a complete constrained graph to an empty graph. We generated 10 instances for each parameter configuration. See Table 6.1.4 for details.

Table 1. Random Max-Clique instances with 150 variables and varying the number of edges ($\%Connectivity$).

Problem	Num. instances	$\%Connectivity$
Max-Clique-150-0	10	0
Max-Clique-150-4	10	17
Max-Clique-150-8	10	35
Max-Clique-150-12	10	52
Max-Clique-150-16	10	70
Max-Clique-150-20	10	87
Max-Clique-150-23	10	100

6.1.5 Hidden Optimal Solution Problems

We also considered the instances with *hidden optimal solutions* presented in [48, 49] and made publicly available by professor Ke Xu [47]. Hereafter, we will refer to the instances of this problem as HOS instances. Observe that these instances are be very difficult to solve by current techniques in spite of their relative small size. The instances are available for different formats such as *Pseudo-Boolean*, *Max-Clique or Minimum Vertex Covering Dimacs Graph* and *Weighted Max-2-SAT* formats. However, all of them model the same problem instances. See a more detailed description of the different sets of problem instances in Table 6.1.5. Observe that they are divided in sets from frb25 to frb59 and each set contains 5 instances. We discarded frb10 to frb20 because they are too small. HOS instances were submitted to different International Evaluations such as the *Pseudo-Boolean Evaluations* and *Max-SAT Evaluations*. Results indicated that (complete and incomplete) solvers were able to find all optimal solutions up to set frb35 and some solvers reached the optimal solution for some instances of the frb40 and frb45 sets but optimality was not certified. A recent algorithm based on iterated tabu search [31] reached better solutions in all sets, even the optimal one for some instances. But most of the instances remain unsolved.

Table 2. Weighted Max-2-SAT instances with hidden optimal solutions.

Problem	Num. instances	Num. variables
frb25	5	325
frb30	5	450
frb35	5	595
frb40	5	760
frb45	5	945
frb50	5	1150
frb53	5	1272
frb56	5	1400
frb59	5	1534

6.1.6 The Combinatorial Auction Problem

Given a set of n *goods* presented in an auction, the bidders generate m *bids* and each one is constituted by a price for a subset of goods. The *Combinatorial Auction* Problem consists in accepting a subset of bids such that the benefits are maximized. It is also a NP-Hard problem. Note that the same good can appear in different bids. Hence, only a bid containing each good can be accepted. We propose the following schema in order to represent this problem as a Max-SAT instance:

- We create a Boolean variable for each bid. Hence, we need m Boolean variables. If we assign to true a Boolean variable x_i, it means that bid i has been accepted. Otherwise, the bid i is not selected.
- For each variable x_i related to the bid i, we create a clause $(\bar{x}_i, w_i)$ where w_i is the price proposed by the bidder. This clause indicates that if this clause is not selected, then the benefit of its price is lost.
- We add a clause $(x_i \vee x_j, \top)$ for each pair of bids sharing some good. This clause avoids selecting the two bids at the same time.

Let M be the sum of the prices of all the bids. The benefit of the combinatorial auction is equivalent to $M - S$, where S is the cost of the optimal solution. We created combinatorial auction instances using the CATS (Combinatorial Auction Test Suite) free generator [21].

It is able to generate problems with three different distributions: *Paths*, *Scheduling* and *Regions*. Each one represents the particular distribution of a real life problem. More information about the combinatorial auction problem can be found in [14]. We generated instances with a fixed number of goods and varying the number of bids. In particular, the number of goods is fixed to 60 and the number of bids varies from 100 to 120 (step 10) for each one of the three distributions. We generated 10 instances for each parameter combination.

6.1.7 The Max-One Problem

Given a satisfiable CNF formula, *Max-One* is the problem of finding a model with a maximum number of variables set to true. This problem can be encoded as Max-SAT by considering the clauses in the original formula as mandatory and adding a weighted unary clause $(x_i, 1)$ for each variable in the formula. Note that solving this problem is much harder than solving the usual SAT problem, because the search cannot stop as soon as a model is found. The optimal model must be found and its optimality must be proved. We consider random 3-SAT instances with 150 variables and the number of clauses ranging from 250 to 550. There are 10 instances for each parameter combination.

6.2 The effect of the Variable Saturation Preprocessor

In this Section the effect of the Variable Saturation Preprocessor is analyzed. In [3], a similar preprocessor was used for systematic search for $K = 6$, 10 and 14. However, no detailed information was given about the selection of the value for the parameter K and the variable selection heuristic. Therefore, preliminary experiments have been performed to determine such parameters. We considered random Max-2-SAT and Max-3-SAT with a fixed number of variables and varying the number of clauses. 10 instances were considered for each parameter configuration and results are presented in plots that show average cpu time in seconds.

The original Max-DP [7, 22], an algorithm completely based on inference, has been used to determine the best heuristic to select the variable to eliminate. Figure 1 compares three different versions of Max-DP depending on the variable selection. The first version selects variables following a lexicographical ordering while the second and third version select variables with maximum and minimum degree, respectively. The experiment considers Max-2-SAT and Max-3-SAT instances with 14 variables and varying the number of clauses. Clearly, the minimum degree ordering is the best for Max-2-SAT and slightly better for Max-3-SAT. Hence, the minimum degree ordering is selected for future experiments.

The best value for K has been determined by exploring the range of possible values from 0 to 20 using the original Max-DP in random Max-2-SAT and Max-3-SAT instances. We observed that for $K = [0 \dots 8]$ very few variables can be eliminated. For $K = [9 \dots 12]$ much more variables can be removed in zero time. For $K > 12$ the time and space required begin to be high. Observe Figure 1 and Figure 2. Clearly, for instances with 14 variables the time begins to be high (Figure 1) while for instances with 16 variables the required time is excessive (Figure 2). Hence, we only considered $K = 12$ and $K = 14$ because they offer a good trade-off between time and space.

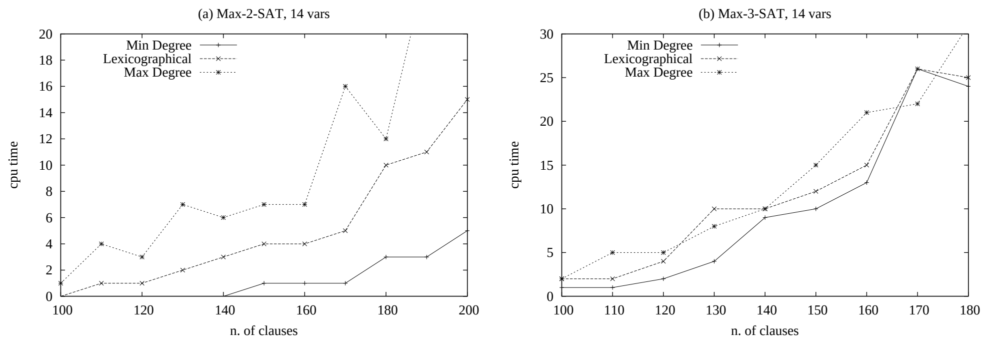

Figure 1. Results on (a) Max-2-SAT and (b) Max-3-SAT instances with 14 variables and varying the number of clauses. Observe the effect of the heuristic used to choose the next variable to be eliminated. The minimum degree heuristic is the best one in both cases.

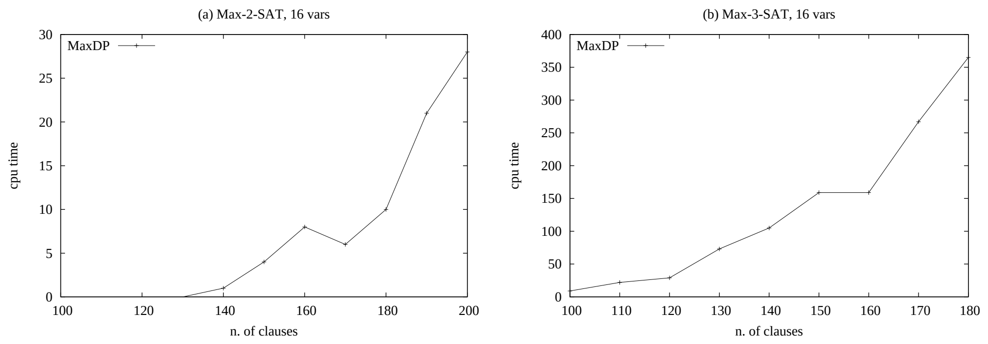

Figure 2. Results on (a) Max-2-SAT and (b) Max-3-SAT instances with 16 variables and varying the number of clauses. Observe that the required cpu time is excessive.

Table 6.2 reports the effect of the Variable Saturation Preprocessor. The table summarizes the number of clauses of the original instance ($Clauses$) and the number of clauses after the preprocessor with $K = 12$ and $K = 14$ ($Clauses_{K=12}$ and $Clauses_{K=14}$, respectively). The number of variables of the original instance ($IniVars$), the number of eliminated variables for each preprocessor ($VE_{K=12}$ and $VE_{K=14}$) and the runtime in seconds ($Time_{K=12}$ and $Time_{K=14}$) are also summarized. The last row shows the *normalized sum*[2.] of the results. The conclusions are detailed next:

- Random Max-Clique, HOS, Scheduling and Regions instances results are omitted. The preprocessor on these instances produces meaningless results where practically no variable is eliminated.
- In general, the total number of eliminated variables is near to the 25% for $K = 12$ and 30% for $K = 14$.

2. The normalized sum is the average of each individual improvement

Table 3. Effect of the Variable Saturation Preprocessor with $K = 12, 14$ on several benchmarks.

Problem	$Clauses$	$Clauses_{K=12}$	$Clauses_{K=14}$	$IniVars$	$VE_{K=12}$	$VE_{K=14}$	$Time_{K=12}$	$Time_{K=14}$
Max-2-SAT-100-300	300	866	2230	100	55	57	0,00	0,00
Max-2-SAT-100-400	400	1474	3340	100	41	43	0,00	0,00
Max-2-SAT-100-500	500	1477	3378	100	31	33	0,00	0,00
Max-2-SAT-100-600	600	2004	4104	100	24	26	0,00	0,10
Max-2-SAT-100-700	700	1876	3636	100	18	22	0,00	0,10
Max-2-SAT-100-800	800	1837	6846	100	11	17	0,00	0,90
Max-2-SAT-100-900	900	1831	5439	100	7	12	0,00	0,50
Max-2-SAT-100-1000	1000	1519	4473	100	4	8	0,00	0,30
Max-Cut-60-200	400	2703	9011	60	28	30	0,00	2,80
Max-Cut-60-300	600	2058	10164	60	16	18	0,00	4,40
Max-Cut-60-400	800	3744	8910	60	10	12	0,00	3,20
Max-Cut-60-500	1000	2223	9236	60	3	6	0,00	3,50
Max-One-150-250	400	534	779	150	63	69	0,00	0,00
Max-One-150-300	450	585	828	150	51	57	0,00	0,00
Max-One-150-350	500	614	742	150	35	42	0,00	0,00
Max-One-150-400	550	638	775	150	26	34	0,00	0,00
Max-One-150-450	600	683	860	150	19	27	0,00	0,00
Max-One-150-500	650	717	832	150	13	19	0,00	0,00
Max-One-150-550	700	737	827	150	6	13	0,00	0,00
Paths-60-100	1046	1600	3208	102	25	29	0,00	0,50
Paths-60-110	1202	2152	3046	102	31	35	0,00	0,00
Paths-60-120	1376	1514	1782	122	28	34	0,00	0,00
Normalized Sums	1,00	2,33	5,95	1,00	0,25	0,30	-	-

- The number of clauses increases excessively in some problems (see Max-Cut instances). In general, we can observe an increment of 133% for $K = 12$ and 495% for $K = 14$.

- The runtime for all problems is practically zero, except for Max-Cut and $K = 14$. The explanation is simple. Some of the eliminated variables for the Max-Cut instances have degree 14 while in the other problem instances most of the eliminated variables have degree smaller than 14.

- In general, $K = 14$ removes more variables but more cpu time is required. Moreover, the resulting problems have significantly more clauses. Differently, $K = 12$ removes slightly less variables than $K = 14$ in zero time and the resulting problems have a smaller number of clauses.

6.3 The effect of the Unit Propagation Preprocessor

This Section presents the effect of the Unit Propagation preprocessor. We considered three values to analyze its effect. The first one is the weight of the empty clause $(\Box, w)$ obtained by the preprocessor. Note that this weight represents a *lower bound* of the optimal solution. The greater its value is, the better we will consider it. The second one is the number of clauses of the resulting problem. The third one is the required time by the preprocessor.

The results of the Unit Propagation preprocessor are reported in Table 6.2. Hereafter, we will refer to Unit Propagation preprocessor as UP. First, the table summarizes the best solution obtained by the Irots algorithm ($IROTS$) [39] and the empty clause obtained after the preprocessor ($(\Box, w)_{UP}$). This comparison helps to identify the proximity of the obtained lower bound from quasi optimal solutions. The table also summarizes the number of clauses of the original and the preprocessed problem ($Clauses$ and $Clauses_{UP}$, respectively), and the runtime in seconds of the preprocessor ($Time_{UP}$). The last row shows the normalized sum of the results. Observe that:

Table 4. The effect of Unit Propagation preprocessor in several benchmarks.

Problem	$IROTS$	$(\Box, w)_{UP}$	$Clauses$	$Clauses_{UP}$	$Time_{UP}$
Max-2-SAT-100-300	14,50	9,80	300,00	326,20	0,00
Max-2-SAT-100-400	28,50	19,50	400,00	423,90	0,00
Max-2-SAT-100-500	43,30	30,50	500,00	532,00	0,00
Max-2-SAT-100-600	60,90	42,70	600,00	612,00	0,00
Max-2-SAT-100-700	77,80	53,10	700,00	715,40	0,00
Max-2-SAT-100-800	96,60	65,50	800,00	805,20	0,00
Max-2-SAT-100-900	112,90	76,70	900,00	890,40	0,00
Max-2-SAT-100-1000	132,30	89,60	1000,00	985,80	0,00
Max-Cut-60-200	47,70	25,40	400,00	349,20	0,00
Max-Cut-60-300	86,30	58,00	600,00	485,80	0,00
Max-Cut-60-400	128,90	90,10	800,00	622,40	0,00
Max-Cut-60-500	172,50	126,90	1000,00	749,60	0,00
Max-One-150-250	20,70	15,40	400,00	400,20	0,00
Max-One-150-300	23,76	16,80	450,00	450,20	0,00
Max-One-150-350	29,05	17,00	500,00	500,00	0,00
Max-One-150-400	32,99	19,20	550,00	550,00	0,00
Max-One-150-450	35,66	18,20	600,00	600,60	0,00
Max-One-150-500	46,15	21,00	650,00	650,00	0,00
Max-One-150-550	48,35	18,90	700,00	700,00	0,00
Max-Clique-150-0	146,80	75,00	10878,00	10803,00	0,02
Max-Clique-150-4	144,40	75,20	9090,00	9014,80	0,01
Max-Clique-150-8	142,21	75,40	7302,00	7226,60	0,01
Max-Clique-150-12	139,44	75,80	5514,10	5438,30	0,01
Max-Clique-150-16	134,60	76,20	3726,00	3649,80	0,00
Max-Clique-150-20	124,21	77,60	1938,00	1862,00	0,00
Max-Clique-150-23	91,34	76,90	597,00	550,30	0,00
Frb25	302,86	163,00	10893,40	10730,40	0,02
Frb30	424,35	225,60	18352,60	18127,00	0,04
Frb35	566,01	298,00	28614,20	28316,20	0,06
Frb40	727,86	381,00	42242,20	41861,20	0,10
Frb45	909,75	473,20	59740,00	59266,80	0,14
Frb50	1111,23	575,60	81795,20	81219,60	0,21
Frb53	1231,30	636,00	95719,40	95083,40	0,25
Frb56	1357,21	701,20	111267,60	110566,40	0,29
Frb59	1489,07	767,00	128147,40	127380,40	0,34
Paths-60-100	65719,27	42038,20	1046,20	1052,20	0,00
Paths-60-110	68000,36	43485,00	1201,50	1206,10	0,00
Paths-60-120	75687,10	48022,20	1375,60	1375,90	0,00
Regions-60-100	69953,29	36769,00	3337,10	3339,60	0,00
Regions-60-110	79149,61	41329,50	3948,00	3949,90	0,00
Regions-60-120	87063,68	45279,70	4933,70	4936,00	0,01
Sched-60-100	88128,90	45537,60	3895,50	3890,10	0,00
Sched-60-110	72748,70	37728,00	4514,50	4509,50	0,00
Sched-60-120	116794,20	60580,20	5451,60	5447,30	0,01
Normalized Sums	1,00	0,59	1,00	0,98	-

- The UP preprocessor can be applied to all problem instances. In the previous Section, we observed that several problem instances remained unmodified within the Variable Saturation preprocessor.

- The empty clause is remarkably high for all problem instances. The UP preprocessor reaches a good lower bound for all instances and they are fairly close to quasi optimal solutions (near to 59%). Recall that the weight of the empty clause in all the original instances is zero.

- The number of clauses after the preprocessor is slightly smaller compared to the original ones (2% smaller). The number of clauses only increases for some instances of the Max-2-SAT problem.

- The required time by the preprocessor is zero in all instances except for the HOS instances.

Table 5. The effect of Clique preprocessor in several benchmarks.

Problem	$Clauses$	$Clauses_{Cl+}$	$Clauses_{Cl-}$	$IROTS$	$(\Box, w)_{Cl+}$	$(\Box, w)_{Cl-}$	$Time_{Cl+}$	$Time_{Cl-}$
Max-Clique-150-0	10878,00	10736,70	10742,00	146,80	143,30	138,00	0,07	0,07
Max-Clique-150-4	9090,00	8957,70	8957,90	144,40	134,30	134,10	0,05	0,05
Max-Clique-150-8	7302,00	7177,40	7177,50	142,21	126,60	126,50	0,04	0,04
Max-Clique-150-12	5514,10	5396,70	5397,90	139,44	119,40	118,20	0,03	0,03
Max-Clique-150-16	3726,00	3617,90	3618,80	134,60	110,10	109,20	0,02	0,02
Max-Clique-150-20	1938,00	1843,40	1845,40	124,21	96,60	94,60	0,00	0,00
Max-Clique-150-23	597,00	524,90	525,40	91,34	74,10	73,60	0,00	0,00
Frb25	10893,40	10595,40	10633,40	302,86	300,00	262,00	0,06	0,07
Frb30	18352,60	17934,60	17989,60	424,35	420,00	365,00	0,11	0,13
Frb35	28614,20	28056,20	28131,00	566,01	560,00	485,20	0,19	0,20
Frb40	42242,20	41524,20	41616,60	727,86	720,00	627,60	0,28	0,31
Frb45	59740,00	58842,00	58953,60	909,75	900,00	788,40	0,41	0,44
Frb50	81795,20	80697,20	80836,80	1111,23	1100,00	960,40	0,57	0,62
Frb53	95719,40	94502,40	94655,60	1231,30	1219,00	1065,80	0,67	0,72
Frb56	111267,60	109925,60	110093,80	1357,21	1344,00	1175,80	0,78	0,85
Frb59	128147,40	126674,40	126853,40	1489,07	1475,00	1296,00	0,90	0,98
Paths-60-100	1046,20	1712,60	4993,60	65719,27	63381,20	64047,20	0,59	9,52
Paths-60-110	1201,50	2248,60	5341,50	68000,36	65331,60	65715,40	0,64	6,17
Paths-60-120	1375,60	2186,20	6466,80	75687,10	71878,00	72953,50	0,67	8,86
Regions-60-100	3337,10	5375,70	6620,70	69953,29	69178,80	69598,90	6,69	115,45
Regions-60-110	3948,00	5737,80	7162,90	79149,61	78244,50	78702,10	8,30	150,94
Regions-60-120	4933,70	6932,50	8061,90	87063,68	86237,30	86666,40	12,98	182,09
Sched-60-100	3895,50	3782,50	5837,70	88128,90	88128,90	88127,80	0,02	90,76
Sched-60-110	4514,50	4389,10	6417,80	72748,70	72680,50	72665,90	0,02	60,16
Sched-60-120	5451,60	5307,70	8796,20	116794,20	116752,90	116560,20	0,03	178,94
Normalized Sums	1,00	1,12	1,58	1,00	0,95	0,91	-	-

6.4 The effect of the Clique Preprocessor

The performance of the Clique preprocessor in several sets of instances is presented in this Section. Hereafter, we will refer to the Clique preprocessor as Cl. Observe that the Cl preprocessor stores all clauses of the weighted formula in a clause list. We considered two variants of Cl based on the position where the resolvents are added in the clause list (line 11 of Algorithm 4 in Section 4). Note that the ordering in which inference is applied can produce very different results [15, 51]. The first one is called $Cl+$ where the resolvents are added at the beginning of the clause list. The second one is called $Cl-$ where the resolvents are added at the end of the clause list. Note that the $Cl-$ strategy was used in [16]. Surprisingly, we have realized empirically that the $Cl+$ is also a good strategy. Other strategies can be explored. However, we only considered these two antagonistic strategies for simplicity.

Table 6.3 reports the results. The table summarizes the number of clauses of the original instance ($Clause$), the number of clauses of each problem set after the $Cl+$ and $Cl-$ preprocessors ($Clauses_{Cl+}$ and $Clauses_{Cl-}$), the weight of the empty clause ($(\Box, w)_{Cl+}$ and $(\Box, w)_{Cl-}$) compared with the best solution of the Irots local search algorithm ($IROTS$), and the runtime in seconds ($Time_{Cl+}$ and $Time_{Cl-}$). The last row shows the normalized sums. Observe that:

- Clique preprocessors only obtain empty clauses in binary unate covering problems. Depending on the structure of the problem instance, the condition to apply the Unit and Star Rule never occurs. Hence, it has no effect on Max-2-SAT, Max-Cut and Max-One instances.
- Observe that the required time for Regions and Scheduling $Cl-$ preprocessed instances explodes. Similarly, the number of clauses generated by $Cl-$ is larger than $Cl+$. Note that soft clauses contained in such original instances have very high weights.

Therefore, we have a situation where the ordering in which inference is applied in $Cl+$ and $Cl-$ produces very different results.

- The weight of the empty clause obtained by Cl preprocessors is much better than those obtained by the UP preprocessor (See Table 6.2). In particular, they are near to 95% for $Cl+$ and 91% for $Cl-$. instances.
- There is a significant increment in the number of clauses (from 12% up to 58%).
- The main drawback is the runtime, specially for $Cl-$. $Cl+$ offers a better trade-off than $Cl-$ between the quality of the empty clause and the runtime.
- In general, the UP preprocessor obtains a good empty clause in negligible time while Clique preprocessors obtain much better empty clause in binary unate covering problems but they require significantly more time and clauses in some problem instances.
- The weight of the empty clauses obtained by $Cl+$ for all HOS and some Scheduling instances is equal to their optimal solution. This observation will become relevant in Section 7.

6.5 Local Search Performance

In this Section we study how preprocessors affect the efficiency of stochastic local search solvers. We considered the following algorithms which apply most of the state-of-the-art techniques in the literature:

- Gsat (GSAT): It is a best improvement algorithm introduced in [38]. Variable selection in Gsat and most of its variants are based on the *score* of the variable under the current assignment. The score is defined as the difference between the number of unsatisfied clauses by the assignment obtained by flipping a variable and the number of unsatisfied clauses by the current assignment.
- Irots (IROTS): Iterated Robust Tabu Search [39]. Irots uses the Rots tabu search presented in [41]. Rots is a best improvement algorithm powered with tabu lists. Irots applies the Rots algorithm in the *search* and *perturbation* phases of an *iterated local search.*
- Walksat (WSAT): It was introduced in [37]. In order to choose a variable to flip, it selects a literal of a randomly chosen unsatisfied clause by the current assignment. It also applies random flips from time to time in order to avoid local minimas.
- Adaptnovelty+ (AN+): Adaptive Novelty+ [19]. It is based on the Walksat architecture and it uses a sophisticated mechanism to avoid local minima. Basically, it consists in applying a larger number of random assignments when a local minima is reached.
- Saps (SAPS): Scaling and Probabilistic Smoothing [43]. It is based on Dynamic Local Search. It alternates a search phase in which variables appearing on unsatisfied clauses are flipped and a phase in which evaluation functions are updated. The key idea is to

Table 6. Results for the Variable Saturation Preprocessor with $K = 12, 14$.

Problem	$GSAT$	$GSAT_{K=12}$	$GSAT_{K=14}$	$IROTS$	$IROTS_{K=12}$	$IROTS_{K=14}$
Max-2-SAT-100-300	14,85	14,84	14,78	14,50	14,50	14,50
Max-2-SAT-100-400	28,75	28,70	28,72	28,50	28,50	28,50
Max-2-SAT-100-500	43,75	43,73	43,64	43,30	43,30	43,30
Max-2-SAT-100-600	61,44	61,34	61,37	60,90	60,90	60,90
Max-2-SAT-100-700	79,10	79,13	79,20	77,80	77,80	77,80
Max-2-SAT-100-800	97,95	97,83	97,92	96,60	96,60	96,60
Max-2-SAT-100-900	114,02	113,96	114,01	112,90	112,90	112,90
Max-2-SAT-100-1000	134,08	134,00	133,91	132,30	132,30	132,30
Max-Cut-60-200	51,02	49,33	49,39	47,70	47,70	47,70
Max-Cut-60-300	90,40	89,33	89,20	86,30	86,30	86,30
Max-Cut-60-400	133,28	132,54	132,38	128,90	128,90	128,90
Max-Cut-60-500	177,18	177,03	176,99	172,50	172,50	172,50
Max-One-150-250	**22,75**	29,53	24,93	20,70	20,70	20,70
Max-One-150-300	**33,13**	70,01	55,88	23,76	23,60	23,60
Max-One-150-350	**80,26**	280,00	230,25	29,05	28,90	28,90
Max-One-150-400	**210,86**	777,21	613,84	32,99	32,90	32,90
Max-One-150-450	**400,79**	1118,73	1067,82	35,66	35,54	35,51
Max-One-150-500	**1368,93**	2474,59	2529,45	46,15	46,02	45,96
Max-One-150-550	**2006,91**	2746,18	3105,31	48,35	48,26	48,22
Paths-60-100	68022,19	66747,07	**66590,17**	65719,27	65700,70	65700,70
Paths-60-110	70375,10	69045,35	**68851,04**	68000,36	67906,90	67906,90
Paths-60-120	78134,82	77065,73	**76778,35**	75687,10	75555,31	75555,30
Normalized Sums	1,0	1,43	1,34	1,00	1,00	1,00

modify evaluation functions in order to prevent the search from getting stuck in local minima.

We compared the efficiency of the above algorithms using the original problem instances and the instances resulting from the three different preprocessors. The five algorithms are implemented in Ubcsat [42] following their original form. Hence, we used the algorithms of Ubcsat in all experiments with default parameters (the default Ubcsat's *cut-off* is 100000 iterations). Local search algorithms are executed 100 times for each problem instance and results are average values. The performance of each local search algorithm is compared with its own performance when the algorithm is fed with the original and preprocessed instances.

Comparisons are presented in two tables with a similar structure to the results in [3, 2]:

- The first table reports results of the best solution found. The first column shows the name of the set of problem instances. The table summarizes the solution obtained by each local search algorithm with the original and the preprocessed instances. The best result for each local search solver is emphasized in bold text. The last row shows the normalized sum of the results. We emphasize best results for each local search solver in bold text, while similar (but different) performances are not emphasized.

- The second table shows the runtime in seconds required by each algorithm to solve the original and the preprocessed instances. Here, each column summarizes the average time in seconds of each set of problem instances and the last one contains the average time needed for all problems. Each row reports the average time for each local search algorithm with the original and preprocessed instances.

6.5.1 Variable Saturation Preprocessor.

Tables 6.5 and 6.5 present the solutions obtained by each local search algorithm within original and $K = 12$ and $K = 14$ preprocessed instances using the Variable Saturation

Table 7. Results for the Variable Saturation Preprocessor with $K = 12, 14$.

Problem	$WSAT$	$WSAT_{K=12}$	$WSAT_{K=14}$	$SAPS$	$SAPS_{K=12}$	$SAPS_{K=14}$	$AN+$	$AN+_{K=12}$	$AN+_{K=14}$
Max-2-SAT-100-300	14,86	14,56	14,56	15,02	14,90	14,87	14,56	14,50	14,50
Max-2-SAT-100-400	30,88	29,89	29,84	29,41	29,16	29,14	29,24	28,59	28,55
Max-2-SAT-100-500	48,48	46,91	46,78	44,38	44,13	44,07	44,87	43,84	43,65
Max-2-SAT-100-600	69,07	67,13	66,91	62,24	61,95	61,89	64,16	62,48	62,18
Max-2-SAT-100-700	89,37	87,46	87,20	79,76	79,67	79,67	82,53	80,87	80,41
Max-2-SAT-100-800	110,69	109,52	108,50	98,66	98,64	98,41	102,89	101,64	100,63
Max-2-SAT-100-900	128,82	127,58	126,69	114,49	114,60	114,55	119,52	118,53	117,61
Max-2-SAT-100-1000	151,13	150,46	149,72	134,79	134,76	134,83	140,82	140,25	139,17
Max-Cut-60-200	51,79	48,39	48,19	47,70	47,80	47,78	48,92	47,70	47,70
Max-Cut-60-300	94,84	91,16	90,59	86,33	86,73	86,74	89,40	87,12	86,77
Max-Cut-60-400	140,72	137,50	136,88	128,97	129,56	129,51	133,70	131,46	130,88
Max-Cut-60-500	186,84	185,74	184,81	172,65	173,50	173,57	179,05	178,12	176,85
Max-One-150-250	21,35	21,03	21,07	23,67	23,31	22,79	20,72	20,70	20,70
Max-One-150-300	24,48	24,34	24,35	29,38	27,29	26,81	23,77	23,66	23,66
Max-One-150-350	30,46	30,32	30,38	38,56	35,38	34,84	29,11	29,05	29,02
Max-One-150-400	34,69	34,44	34,52	52,57	43,02	41,07	33,08	32,98	32,92
Max-One-150-450	37,27	37,40	37,40	93,91	67,65	59,98	35,77	35,76	35,72
Max-One-150-500	47,29	47,20	47,23	422,65	353,07	280,92	46,17	46,12	46,07
Max-One-150-550	49,31	49,30	49,27	731,48	710,73	664,50	48,55	48,54	48,45
Paths-60-100	68604	66448	**66207**	66884	66399	**66277**	66391	65817	**65762**
Paths-60-110	71289	68687	**68493**	69130	68792	**68674**	68778	68025	**67982**
Paths-60-120	79241	76658	**76303**	76883	76512	**76342**	76405	75740	**75664**
Normalized Sums	1,00	0,98	0,98	1,00	0,96	0,94	1,00	0,99	0,99

preprocessor. Table 6.5 presents results for Gsat and Irots and Table 6.5 shows results for Walksat, Saps and Adaptnovelty+. Table 6.5.1 shows the runtime. We can observe the next conclusions:

- All algorithms except Irots, report a noticeable improvement in Paths instances. Irots does not report significant improvement nor worsening.
- In general, slight improvements are detected for Walksat, Adaptnovelty+ and Saps.
- Gsat shows a global worsening of approximately 40%. The reason is because of the worse results obtained for Max-One instances, while for the other instances a slight improvement is noticed.
- The original instances can be solved in less time than the preprocessed ones. Moreover, the runtime highly depends on the value of K. In particular, a larger K implies more clauses which imply more runtime.

6.5.2 Unit Propagation Preprocessor.

Table 6.5.2 compares the solutions obtained by each local search algorithm within the original and UP-preprocessed instances. Table 6.5.2 shows the cpu time in seconds. We observe that:

- All algorithms except Irots improve their performance. The global improvement for Gsat, Walksat, Saps and Adaptnovelty+ are 6%, 3%, 5% and 1%, respectively. No algorithm reports improvement on Scheduling instances.
- In general, the UP preprocessor helps the Walksat and Adaptovelty+ algorithms to obtain a significant improvement in all problem instances. Improvements are observed for Gsat and Saps in Max-One instances.

Table 8. Runtime averages for the Variable Saturation Preprocessor with $K = 12, 14$.

SLS / Problem	Max-2-SAT	Max-Cut	Max-One	Paths	Average
$GSAT$	0,08	0,11	0,15	0,08	**0,11**
$GSAT_{K=12}$	0,23	0,52	0,12	0,34	0,30
$GSAT_{K=14}$	0,54	1,91	0,14	0,64	0,80
IROTS	0,12	0,12	0,21	0,15	**0,15**
$IROTS_{K=12}$	0,30	0,60	0,19	0,36	0,36
$IROTS_{K=14}$	0,60	2,02	0,20	0,71	0,88
$WSAT$	0,05	0,07	0,06	0,07	**0,06**
$WSAT_{K=12}$	0,19	0,43	0,07	0,15	0,21
$WSAT_{K=14}$	0,45	1,62	0,09	0,28	0,61
$SAPS$	0,15	0,19	0,13	0,23	**0,17**
$SAPS_{K=12}$	0,29	0,67	0,15	0,32	0,36
$SAPS_{K=14}$	0,66	2,23	0,18	0,44	0,88
$AN{+}$	0,06	0,08	0,07	0,08	**0,07**
$AN{+}_{K=12}$	0,25	0,59	0,09	0,21	0,29
$AN{+}_{K=14}$	0,62	2,30	0,10	0,40	0,86

Table 9. Results for the Unit Propagation Preprocessor.

Problem	$GSAT$	$GSAT_{UP}$	$IROTS$	$IROTS_{UP}$	$WSAT$	$WSAT_{UP}$	$SAPS$	$SAPS_{UP}$	$AN{+}$	$AN{+}_{UP}$
Max-2-SAT-100-300	14.85	14.84	14.50	14.50	14.86	14.50	15.02	14.71	14.56	14.50
Max-2-SAT-100-400	28.75	28.75	28.50	28.50	30.88	**28.50**	29.41	29.15	29.24	**28.50**
Max-2-SAT-100-500	43.75	43.71	43.30	43.30	48.48	**43.44**	44.38	43.85	44.87	**43.31**
Max-2-SAT-100-600	61.44	61.48	60.90	60.90	69.07	**61.66**	62.24	61.84	64.16	**60.97**
Max-2-SAT-100-700	79.10	79.17	77.80	77.80	89.37	**79.97**	79.76	79.16	82.53	**78.00**
Max-2-SAT-100-800	97.95	97.99	96.60	96.60	110.69	**100.23**	98.66	98.01	102.89	**97.16**
Max-2-SAT-100-900	114.02	114.09	112.90	112.90	128.82	**116.72**	114.49	114.05	119.52	**113.49**
Max-2-SAT-100-1000	134.08	133.95	132.30	132.30	151.13	**137.65**	134.79	133.95	140.82	**133.71**
Max-Cut-60-200	51.02	51.16	47.70	47.70	51.79	**48.24**	47.70	47.79	48.92	47.71
Max-Cut-60-300	90.40	90.41	86.30	86.30	94.84	**87.51**	86.33	86.60	89.40	**86.33**
Max-Cut-60-400	133.28	133.22	128.90	128.90	140.72	**131.45**	128.97	129.41	133.70	**129.23**
Max-Cut-60-500	177.18	177.46	172.50	172.50	186.84	**176.08**	172.65	173.17	179.05	**172.74**
Max-One-150-250	22.75	22.88	20.70	20.70	21.35	**20.70**	23.67	**21.19**	20.72	20.75
Max-One-150-300	**33.13**	35.44	23.76	23.75	24.48	**23.64**	29.38	**25.70**	23.77	23.62
Max-One-150-350	80.26	**59.87**	29.05	29.04	30.46	**28.96**	38.56	**34.45**	29.11	28.90
Max-One-150-400	210.86	**125.33**	32.99	32.98	34.69	**33.05**	52.57	**43.40**	33.08	32.91
Max-One-150-450	400.79	**199.76**	35.66	35.65	37.27	**35.77**	93.91	**62.33**	35.77	35.57
Max-One-150-500	1368.93	**446.96**	46.15	46.11	47.29	**46.39**	422.65	**160.20**	46.17	45.96
Max-One-150-550	2006.91	**590.32**	48.35	48.34	49.31	**48.81**	731.48	**240.10**	48.55	48.36
Max-Clique-150-0	146,94	146,954	146,80	146,8	147,00	146,987	147,04	147,014	146,96	146,92
Max-Clique-150-4	144,94	144,945	144,40	144,4	145,95	145,63	145,25	145,225	145,50	145,07
Max-Clique-150-8	142,98	142,976	142,21	142,214	144,81	144,032	143,35	143,419	143,79	143,39
Max-Clique-150-14	140,22	140,187	139,44	139,447	143,05	**142,098**	140,76	140,783	141,41	**140,79**
Max-Clique-150-16	135,41	135,435	134,60	134,639	140,49	**138,737**	136,30	136,414	137,39	**136,34**
Max-Clique-150-20	124,39	124,40	124,21	124,25	133,80	**129,654**	126,19	126,43	127,87	**125,40**
Max-Clique-150-23	90,00	90,65	91,34	91,31	104,65	**90,694**	92,90	93,61	94,04	**89,87**
Frb25	302.08	302.03	302.86	302.84	311.74	**309.70**	305.00	304.98	305.55	304.24
Frb30	422.63	422.61	424.35	424.34	435.38	**433.00**	426.60	426.20	426.65	425.03
Frb35	563.05	563.13	566.01	565.99	578.84	**576.09**	568.09	567.83	567.48	566.00
Frb40	723.54	723.59	727.86	727.90	742.63	**739.57**	729.30	729.20	728.39	**726.75**
Frb45	904.14	904.19	909.75	909.70	926.41	**923.09**	911.12	910.73	909.26	**907.55**
Frb50	1104.63	1104.63	1111.23	1111.24	1130.19	**1126.60**	1112.21	1112.18	1110.19	**1108.33**
Frb56	1349.41	1349.35	1357.21	1357.19	1378.77	**1374.81**	1358.65	1358.51	1355.17	**1353.54**
Frb59	1480.59	1480.65	1489.07	1489.14	1511.97	**1507.88**	1490.43	1490.44	1486.90	1485.04
Paths-60-100	68022	68004	65719	65718	68604	**66637**	**66884**	66928	66391	**65799**
Paths-60-110	70375	70385	68000	68001	71289	**69235**	**69130**	69297	68778	**68074**
Paths-60-120	78134	78199	75687	75689	79241	**77088**	**76883**	76957	76405	**75778**
Regions-60-100	70779	70800	69953	69948	70169	**70080**	70604	70608	70031	**69975**
Regions-60-110	79861	79858	79149	79150	79496	**79408**	79676	79673	79316	**79212**
Regions-60-120	87871	87876	87063	87062	87354	**87281**	87678	87683	87219	**87139**
Sched-60-100	88171	88171	88128	88128	88128	88128	88218	88220	88128	88128
Sched-60-110	72860	72855	72748	72748	72748	72748	72932	72918	72748	72748
Sched-60-120	116933	116944	116794	116794	116794	116794	116884	116873	116794	116794
Normalized Sums	1.00	0.94	1.00	1.00	1.00	0.97	1.00	0.95	1.00	0.99

- The original and preprocessed instances require similar times to be solved.

Compared to the Variable Saturation Preprocessor, the UP Preprocessor globally obtains better results for all local search algorithms. Only for the Paths instances, the Variable Saturation Preprocessor reports better results.

Table 10. Runtime averages for the Unit Propagation Preprocessor.

SLS / Problem	Max-2-SAT	Max-Cut-60	Max-One-150	Paths-60	Regions-60	Sched-60	Max-Clique-150	Frb	Average
$GSAT$	0,08	0,11	0,15	0,08	0,09	0,25	0,31	1,21	0,28
$GSAT_{UP}$	0,12	0,11	0,15	0,08	0,09	0,26	0,31	1,23	0,29
$IROTS$	0,12	0,12	0,21	0,15	0,25	0,33	0,39	1,70	0,41
$IROTS_{UP}$	0,19	0,17	0,12	0,16	0,25	0,34	0,39	1,72	0,43
$WSAT$	0,05	0,07	0,06	0,07	0,10	0,10	0,11	0,36	0,11
$WSAT_{UP}$	0,09	0,10	0,06	0,08	0,12	0,11	0,12	0,28	0,12
$SAPS$	0,15	0,19	0,13	0,23	0,64	0,61	0,69	2,90	0,69
$SAPS_{UP}$	0,14	0,16	0,11	0,18	0,52	0,51	0,57	2,38	0,57
$AN+$	0,06	0,08	0,07	0,08	0,13	0,11	0,12	0,48	0,14
$AN+_{UP}$	0,11	0,12	0,07	0,10	0,16	0,14	0,15	0,36	0,15

Table 11. Results for Clique Preprocessors.

Problem	$GSAT$	$GSAT_{Cl+}$	$GSAT_{Cl-}$	$IROTS$	$IROTS_{Cl+}$	$IROTS_{Cl-}$
Max-Clique-150-0	146,94	146,97	146,94	146,80	146,80	146,80
Max-Clique-150-4	144,94	144,95	144,95	144,40	144,40	144,40
Max-Clique-150-8	142,98	142,99	142,98	142,21	142,21	142,22
Max-Clique-150-12	140,22	140,19	140,18	139,44	139,44	139,45
Max-Clique-150-16	135,41	135,40	135,43	134,60	134,61	134,63
Max-Clique-150-20	124,39	124,39	124,37	124,21	124,20	124,21
Max-Clique-150-23	90,00	90,03	89,99	91,34	91,39	91,38
Frb25	302,08	302,15	302,10	302,86	302,80	302,84
Frb30	422,63	422,61	422,64	424,35	424,32	424,36
Frb35	563,05	563,02	563,10	566,01	565,96	565,97
Frb40	723,54	723,55	723,54	727,86	727,76	727,82
Frb45	904,14	904,06	904,19	909,75	909,56	909,57
Frb50	1104,63	1104,54	1104,55	1111,23	1111,09	1111,19
Frb53	1224,01	1224,11	1224,02	1231,30	1231,27	1231,37
Frb56	1349,41	1349,29	1349,36	1357,21	1357,13	1357,12
Frb59	1480,59	1480,62	1480,64	1489,07	1488,95	1489,11
Paths-60-100	68022,19	68086,51	68030,27	65719,27	65716,88	65718,77
Paths-60-110	70375,10	70358,06	70437,80	68000,36	67992,62	67991,29
Paths-60-120	78134,82	78138,46	78238,26	75687,10	75682,97	75690,01
Regions-60-100	70779,73	70810,74	70791,79	69953,29	69950,76	69949,46
Regions-60-110	79861,86	79868,43	79833,42	79149,61	79148,47	79144,63
Regions-60-120	87871,64	87896,96	87866,13	87063,68	87062,49	87062,58
Sched-60-100	88171,99	88173,34	88179,33	88128,90	88128,90	88128,90
Sched-60-110	72860,04	72846,34	72881,83	72748,70	72748,70	72748,70
Sched-60-120	116933,73	116956,10	116956,00	116794,20	116794,20	116794,20
Normalized Sums	1,00	1,00	1,00	1,00	1,00	1,00

6.5.3 Clique Preprocessor.

Tables 6.5.2 and 6.5.2 present the obtained solutions by each local search algorithm within original and $Cl+$ and $Cl-$ preprocessed instances. Table 6.5.2 shows results for Gsat and Irots and Table 6.5.2 reports the results for Walksat, Saps and Adaptnovelty+. Table 6.5.3 shows the runtime. We can conclude that:

- No significant improvement is obtained with Gsat and Irots. On the contrary, both $Cl+$ and $Cl-$ preprocessed instances improve the performance of the other algorithms.

- Considering Walksat and Adaptnovelty+ algorithms, $Cl+$ produces the best results for HOS and Max-Clique instances, while $Cl-$ reports slightly better results on Paths and Regions instances. The same behaviour is observed for Saps with the only difference that no improvement is found for Max-Clique instances.

- All algorithms require similar or even less runtime for Max-Clique, HOS and Scheduling instances using the $Cl+$ variant.

Table 12. Results for Clique Preprocessors.

Problem	$WSAT$	$WSAT_{Cl+}$	$WSAT_{Cl-}$	$SAPS$	$SAPS_{Cl+}$	$SAPS_{Cl-}$	$AN+$	$AN+_{Cl+}$	$AN+_{Cl-}$
Max-Clique-150-0	147,00	146,80	146,80	147,04	146,83	147,06	146,96	146,80	146,80
Max-Clique-150-4	145,95	144,49	144,52	145,25	145,35	145,34	145,50	**144,42**	144,43
Max-Clique-150-8	144,81	**142,89**	142,91	143,35	143,49	143,50	143,79	**142,64**	142,65
Max-Clique-150-12	143,05	**140,68**	140,76	140,76	140,87	140,90	141,41	**139,89**	139,94
Max-Clique-150-16	140,49	**136,84**	136,92	136,30	136,58	136,49	137,39	**135,40**	135,48
Max-Clique-150-20	133,80	**127,37**	127,61	126,19	126,58	126,59	127,87	**124,31**	124,51
Max-Clique-150-23	104,65	**91,07**	91,13	92,90	93,80	93,92	94,04	89,99	**89,98**
Frb25	311,74	**300,31**	306,33	305,00	**300,06**	304,71	305,55	**300,06**	302,77
Frb30	435,38	**420,90**	428,96	426,60	**420,31**	426,00	426,65	**420,22**	423,32
Frb35	578,84	**561,84**	571,59	568,09	**561,20**	567,51	567,48	**560,91**	563,87
Frb40	742,63	**722,62**	734,22	729,30	**721,71**	728,77	728,39	**721,29**	724,27
Frb45	926,41	**903,66**	916,92	911,12	**902,52**	910,42	909,26	**901,89**	904,80
Frb50	1130,19	**1104,70**	1119,92	1112,21	**1103,29**	1111,51	1110,19	**1102,28**	1105,21
Frb53	1251,50	**1224,61**	1240,77	1232,76	**1223,23**	1232,15	1229,90	**1221,79**	1224,69
Frb56	1378,77	**1350,34**	1367,36	1358,65	**1348,80**	1357,88	1355,17	**1347,03**	1350,05
Frb59	1511,97	**1482,02**	1499,95	1490,43	**1480,69**	1489,90	1486,90	**1478,21**	1481,18
Paths-60-100	68604	65700	**65700**	66884	66328	**66183**	66391	**65700**	65700
Paths-60-110	71289	67913	**67907**	69130	68748	**68616**	68778	67919	**67907**
Paths-60-120	79241	75578	**75558**	76883	76763	**76509**	76405	75584	**75556**
Regions-60-100	70169	69938	**69938**	70604	70163	**70013**	70031	**69940**	69946
Regions-60-110	79496	79127	**79126**	79676	79303	**79206**	79316	79137	**79132**
Regions-60-120	87354	**87040**	87041	87678	87240	**87090**	87219	87063	**87061**
Sched-60-100	88128,90	88128,90	88128,90	88218,19	88128,90	88128,90	88128,90	88128,90	88128,90
Sched-60-110	72748,70	72748,70	72748,70	72932,03	72748,70	72748,70	72748,70	72748,70	72748,70
Sched-60-120	116794,20	116794,20	116794,20	116884,32	116794,20	116794,20	116794,20	116794,20	116794,20
Normalized Sums	1,00	0,97	0,98	1,00	1,00	1,00	1,00	0,99	0,99

Table 13. Runtime averages for Clique Preprocessors.

SLS / Problem	Max-Clique-150	Frb	Paths-60	Regions-60	Sched-60	Average
$GSAT$	0,31	1,21	0,08	0,09	0,25	0,39
$GSAT_{Cl+}$	0,31	1,09	0,17	0,22	0,08	**0,38**
$GSAT_{Cl-}$	0,31	1,07	0,44	0,37	8,54	2,15
$IROTS$	0,39	1,70	0,15	0,25	0,33	**0,56**
$IROTS_{Cl+}$	0,39	1,62	0,38	3,26	0,03	1,13
$IROTS_{Cl-}$	0,39	1,59	1,31	5,44	2,73	2,29
$WSAT$	0,11	0,36	0,07	0,10	0,10	**0,15**
$WSAT_{Cl+}$	0,18	0,51	0,18	1,24	0,02	0,43
$WSAT_{Cl-}$	0,16	0,29	0,77	2,34	0,81	0,87
$SAPS$	0,69	2,90	0,23	0,64	0,61	1,01
$SAPS_{Cl+}$	0,39	1,11	0,71	2,80	0,02	**1,01**
$SAPS_{Cl-}$	0,40	1,39	1,26	4,89	2,52	2,09
$AN+$	0,12	0,48	0,08	0,13	0,11	**0,18**
$AN+_{Cl+}$	0,21	0,58	0,24	5,59	0,02	1,33
$AN+_{Cl-}$	0,19	0,35	1,22	12,74	3,78	3,66

- In general, all algorithms require much more runtime for $Cl-$ preprocessed instances for Paths, Regions and Scheduling. Besides, all algorithms require more runtime for Paths and Regions using $Cl+$ preprocessed instances. Again, a larger number of clauses on the preprocessed instance imply more cpu time.

- $Cl+$ offers in general better trade-off between the quality of the solutions and runtime than $Cl-$.

- Global results seems to report a small improvement. However, if the best solution of individual problems is compared, Clique preprocessor produces much better results than UP preprocessor for the binary unate covering problem.

Table 14. Best preprocessor for each benchmark and local search algorithm.

Problem / SLS	*GSAT*	*IROTS*	*WSAT*	*SAPS*	*AN+*
Max-2-SAT	-	-	UP	-	UP
Max-Cut-60	VS K=14	-	UP	-	UP
Max-One-150	UP	-	UP	UP	-
Max-Clique-150	-	-	Cl+	-	Cl+
HOS	-	-	Cl+	Cl+	Cl+
Paths-60	VS K=14	VS K=14	Cl-	Cl-	Cl-
Regions-60	-	-	Cl-	Cl-	Cl-
Sched-60	-	-	-	-	-

6.6 Conclusions about the experiments

In this Section, we point out some concluding remarks about the set of experiments. Table 6.6 reports the best preprocessor for each problem and local search algorithm based on the previous experiments. We can conclude that:

- Variable Saturation preprocessor (labelled as VS in the table) reports improvements only for Gsat and Irots and for the Max-Cut and Paths instances. The best variant is with $K = 14$.
- Unit Propagation preprocessor (labelled as UP) reports significant improvement in Max-2-SAT, Max-Cut and Max-One instances for Walksat, Adaptnovelty+ and Saps. In general, better solutions are obtained using the UP preprocessor. Besides, there are no significant differences in the runtime of the SLS algorithms with the original and preprocessed instances.
- Clique Preprocessor (variants labelled with $Cl+$ and $Cl-$) is suitable for the remaining problem instances (except Scheduling). Noticeable improvements for Walksat, Saps and Adaptnovelty+ have been observed.
- No preprocessor could improve the original results of the local search algorithms for Scheduling instances. The reason is simple: Local search algorithms are able to find the optimal solution quite easily within original and preprocessed instances.
- In general, we observe that Gsat and Irots are less sensitive to inference. Differently, Walksat, Saps and Adaptnovelty+ are quite sensitive.
- In general, preprocessors that generate a large number of clauses affect negatively to the performance of SLS algorithms (specially Variable Saturation and $Cl-$).
- After the experiments, we observed that $Cl+$ preprocessor returns optimal solutions for all HOS and several Scheduling instances. In the following Section, we prove empirically that such instances can be solved to optimality with a SLS algorithm.

We can conclude that Variable Saturation Preprocessor does not produce noticeable improvements while Unit Propagation and Clique Preprocessors report significant improvements for some SLS algorithms. While Variable Saturation Preprocessor is not effective in general, it should be very useful for instances in which all variables have very small degree.

Let us define two groups of SLS Algorithm:

Table 15. The average size of the unsatisfied clause list managed by Walksat is clearly different for original and preprocessed instances. This may be the explanation why focused algorithms improve their performance with UP and $Clique$ preprocessors.

Problem	$WSAT$	$WSAT_{UP}$	$WSAT_{Cl+}$
Max-2-SAT-100-1000	189,30	74,10	-
Max-Cut-60-500	225,10	82,40	-
Max-One-150-400	148,00	71,20	-
Max-Clique-150-12	47,00	23,20	26,10
Frb59	111,50	90,80	169,00
Paths-60-120	120,80	116,60	802,40
Regions-60-120	152,70	142,90	403,60
Sched-60-120	1597,20	826,20	97,80

- *Focused Algorithms*: They select the next variable to flip occurring in an *unsatisfied clause*. In particular, they manage a list of the unsatisfied clauses by the current assignment. At each iteration, they randomly select a clause occurring in such list (Walksat and Adaptnovelty+) or a variable occurring in an unsatisfied clause (Saps). Hence, Walksat, Adaptnovelty+ and Saps are focused algorithms.

- *Best Improvement Algorithms*: They select the next variable to flip with maximal score. Gsat and Irots are best improvement algorithms.

Unit Propagation and Clique Preprocessors are specially useful for focused algorithms while they are not so good for best improvement algorithms. The application of both preprocessors produces a powerful empty clause and a different (but equivalent) problem instance. Observe that the contribution of many unsatisfied clauses is contained in the empty clause generated by the preprocessors. Hence, we conjecture that the list of unsatisfied clauses used by focused algorithms is very different when they are fed with original and preprocessed instances.

We have run a new experiment to investigate it. We modified the Ubcsat implementation to compute the average size of the unsatisfied clause list at each search step. Results are presented in Table 6.6. It considers the Walksat algorithm, the UP and $Cl+$ preprocessors and a representative subset of each benchmark considered in this paper. The first column shows the name of the benchmark. The second column shows the average size of the unsatisfied clause list (i.e. the average number of clauses contained in such list) at each search step for the original instances. Similarly, the third and fourth columns report the average size of the unsatisfied clause list at each search step for the UP and $Cl+$ preprocessed instances, respectively. Clearly, the Walksat algorithm manages different sized unsatisfied clause lists with the preprocessed instances and this may lead to more focused and accurate selections. We obtained similar results for Saps and Adaptnovelty+.

7. Proving optimality

As we stated before, $Cl+$ preprocessor obtained an empty clause with a weight equal to the optimal solution for all HOS instances and for some Scheduling instances (sets Sched-60*). For those instances, we carried out a new experiment to prove their optimality. We also considered the combinatorial auction instances with Scheduling distribution included in [24, 21]: There are a total of 500 instances with 144 goods and 1000 bids on each instance. We will refer to them as *1k-144*.

In order to prove optimality, the local search algorithm is executed with a target value equal to the weight of the empty clause (i.e. a lower bound of the optimal solution) generated by the preprocessor. The optimality is proved when the local search algorithm obtains a solution equal to the specified target value within a limit (time or number of search steps) because the lower bound is reached as the optimal solution.

Initially, a set of experiments was performed to determine the most effective local search algorithm in HOS and Scheduling original and *Cl+* preprocessed instances. For Scheduling instances, all algorithms performed fairly well. For HOS instances, Gsat, Irots and Walksat were the worst options. Saps obtained good results but it was clearly improved by Adaptnovelty+ in the most difficult instances. Furthermore, Adaptnovelty+ satisfies the *PAC* (*Probabilistically approximately complete*) property [20]. An incomplete algorithm is PAC if it finds a solution of soluble instances with a probability approaching one as the run-time approaches to infinity. Hence, we have chosen Adaptnovelty+ for the entire experiment.

We also tested the MiniMaxSat [18] systematic search solver in such instances to analyze which instances can be solved by a systematic search in reasonable time (1200 seconds). MiniMaxSat was able to solve all the Sched-60* sets in less than 5 seconds, while it got stuck in most of the HOS from frb40 to frb59 and in 1k-144 instances.

Table 7 summarizes the results of solving HOS instances with the original and preprocessed instances. Adaptnovelty+ was executed with a time limit of 1200 seconds and 10 runs per instance. Column *Problem* indicates the name of each individual instance and column *Optimal* shows the value of its optimal solution. Then, the solutions reached by Adaptnovelty+ with the original instances are presented in column *AN+*. The following two columns represent its success ratio to reach the optimal solution (*%Success*) and the average time (*Time*). Then, the solutions found by Adaptnovelty+ with the preprocessed instances are presented in column $AN+_{Cl+}$. Similarly, its success ratio and average time are shown in columns $\%Success_{Cl+}$ and $Time_{Cl+}$, respectively. The last row shows average results. Note that the preliminary experiments of this Section and the results in Table 7 were conducted on a 3Ghz Intel Xeon computer with 4GB of memory and Linux.

Observe the results for *AN+*: All instances remain unsolved within the time limit. Differently, for $AN+_{Cl+}$ all instances are solved in less than 1200 seconds for at least one of the ten runs except instance frb59-26-2. Adaptnovelty+ was run again for this instance to check if the optimal solution could be found without a time limit. The instance was solved to optimality in 12 hours approximately.

Within our approach, all HOS instances have been solved and their optimality is guaranteed. Hence, we can say that all HOS instances have been solved to optimality using a stochastic local search. Note that these difficult instances were submitted to several International Competitions including the *Pseudo-Boolean Evaluation* and *Max-Sat Evaluation* (See results in [47]). Complete and incomplete solvers were able to reach the optimal solution up to some instances of the frb45 set. The other instances remained unsolved. In the recent work [31], a novel Iterated Tabu Search (ITS) is able to reach the optimal solution for more instances or better sub-optimal solutions than previous works, but 22 instances remained unsolved. Such ITS reaches its best solutions mainly in the first second of execution time

Table 16. Comparison of Adaptnovelty+ for HOSinstances considering original instances and $Cl+$ preprocessed ones. The optimal solution is found and certified for at least one run for all preprocessed instances except one.

Problem	Optimal	$AN+$	$\%Success$	$Time$	$AN+_{Cl+}$	$\%Success_{Cl+}$	$Time_{Cl+}$
frb25-13-1	300	305,00	0	1200,01	300,00	100	0,04
frb25-13-2	300	304,80	0	1200,00	300,00	100	0,13
frb25-13-3	300	304,80	0	1200,01	300,00	100	0,05
frb25-13-4	300	304,70	0	1200,00	300,00	100	0,03
frb25-13-5	300	305,00	0	1200,01	300,00	100	0,02
frb30-15-1	420	426,90	0	1200,00	420,00	100	0,06
frb30-15-2	420	426,00	0	1200,00	420,00	100	0,08
frb30-15-3	420	426,50	0	1200,00	420,00	100	0,41
frb30-15-4	420	426,60	0	1200,00	420,00	100	0,03
frb30-15-5	420	426,20	0	1200,00	420,00	100	0,23
frb35-17-1	560	567,50	0	1200,00	560,00	100	2,00
frb35-17-2	560	568,00	0	1200,00	560,00	100	0,92
frb35-17-3	560	567,50	0	1200,00	560,00	100	0,29
frb35-17-4	560	567,90	0	1200,00	560,00	100	2,39
frb35-17-5	560	567,80	0	1200,00	560,00	100	0,19
frb40-19-1	720	728,60	0	1200,00	720,00	100	0,94
frb40-19-2	720	728,60	0	1200,00	720,00	100	6,98
frb40-19-3	720	727,90	0	1200,00	720,00	100	2,27
frb40-19-4	720	728,40	0	1200,00	720,00	100	10,78
frb40-19-5	720	728,30	0	1200,00	720,00	100	41,94
frb45-21-1	900	909,20	0	1200,00	900,00	100	21,39
frb45-21-2	900	909,10	0	1200,00	900,00	100	23,12
frb45-21-3	900	909,10	0	1200,00	900,00	100	79,07
frb45-21-4	900	909,80	0	1200,00	900,00	100	14,17
frb45-21-5	900	909,20	0	1200,00	900,00	100	110,36
frb50-23-1	1100	1109,70	0	1200,00	1100,00	100	168,79
frb50-23-2	1100	1110,60	0	1200,00	1100,00	100	445,74
frb50-23-3	1100	1110,40	0	1200,00	1100,50	50	1016,30
frb50-23-4	1100	1109,80	0	1200,00	1100,00	100	44,49
frb50-23-5	1100	1111,70	0	1200,00	1100,00	100	39,04
frb53-24-1	1219	1231,00	0	1200,01	1219,80	20	1117,07
frb53-24-2	1219	1227,90	0	1200,00	1219,50	50	833,91
frb53-24-3	1219	1229,20	0	1200,00	1219,00	100	284,16
frb53-24-4	1219	1229,30	0	1200,01	1219,60	40	889,53
frb53-24-5	1219	1229,30	0	1200,01	1219,00	100	253,53
frb56-25-1	1344	1356,30	0	1200,01	1344,90	10	1083,46
frb56-25-2	1344	1354,40	0	1200,01	1344,60	40	847,68
frb56-25-3	1344	1354,70	0	1200,01	1344,50	50	936,87
frb56-25-4	1344	1354,80	0	1200,00	1344,00	100	309,71
frb56-25-5	1344	1355,20	0	1200,01	1344,00	100	133,07
frb59-26-1	1475	1485,20	0	1200,00	1475,90	10	1085,56
frb59-26-2	1475	1487,20	0	1200,00	1476,00	0	1200,00
frb59-26-3	1475	1487,60	0	1200,00	1475,90	10	1148,36
frb59-26-4	1475	1487,40	0	1200,00	1475,90	10	1178,94
frb59-26-5	1475	1486,70	0	1200,00	1475,00	100	135,47
Averages	893,11	902,04	0,00	1200,00	893,29	82,00	299,32

while no improvements are obtained later. Finally, the recent work [8], done in parallel to ours, presents a specific SLS algorithm for the Maximum Clique (and Vertex Covering) Problem which is very effective in the HOS instances. In particular, it is able to reach the optimal solution for all instances. However, both [31, 8] are unable to certify the optimality.

Table 7 shows the results for the Scheduling instances. Adaptnovelty+ was launched with a time limit of 10 seconds and 10 runs per instance since these instances are quite easy to solve for SLS algorithms. In particular, we focus our study on how many instances can be solved to optimality and such optimality is in addition *certified*.

Column Problem refers to the name of the problem set. Column $\#Instances$ refers to the number of instances for each problem set. Column $Time_{Cl+}$ refers to the average time in seconds required by the $Cl+$ preprocessor for all instances of each set. Finally, column *#Optimality* refers to the number of instances that are solved and its optimality is certified. The value inside brackets is the average time in seconds to prove their optimality.

Table 17. Proving optimality for Scheduling instances.

Problem	#Instances	$Time_{Cl+}$	$\#Optimality$
Sched-60-100	10	0,02	10 (0,00)
Sched-60-110	10	0,02	9 (0,00)
Sched-60-120	10	0,03	9 (0,00)
1k-144	500	1,41	118 (0,15)

The optimality is certified for 28 of the 30 instances of the Sched-60* sets and for 118 of the 500 instances of the 1k-144 set. In fact, the optimality is certified for the 10 runs of each instance. Required time by the $Cl+$ preprocessor and to prove optimality are negligible for Sched-60* sets. Required times for the $1k - 144$ set are also small: An average of 1,41 seconds is needed to run the $Cl+$ preprocessor an average of 0,15 seconds is needed to certify optimality.

8. Related Work

The three preprocessors presented in this paper are based on well-known limited inference methods used in systematic search algorithms [18, 16, 3]. Our current work is the first step towards the integration of inference and local search for Max-SAT.

Some relevant works exist about the integration of local search and classical resolution in SAT. Such works preserve the *satisfiability* of the problem instance. That is, if the input instance is satisfiable (unsatisfiable) the resulting instance of a resolution process is satisfiable (unsatisfiable). However, they do not preserve the *equivalence* (See Section 2) from an optimization point of view and cannot be applied directly to the Max-SAT problem. In [9], new clauses are added during the search process. In particular, resolution is applied between a pair of unsatisfied clashing clauses at the local minima. In [13], a local search algorithm is powered with resolution and the resulting algorithm is advocated to be *complete*. Finally, in [2] a preprocessor restricted to clauses of size 3 is presented. The preprocessor is shown to be very effective on *quasigroup existence problems* and some random and real-world SAT problems.

The *Variable Saturation preprocessor* was introduced in [3] and it is a restricted version of a complete inference algorithm for Max-SAT [7, 22]. The work in [3] showed that systematic search solvers boosted their performance in some problem instances after applying the preprocessor. However, limited information is detailed about the setting of some parameters of the preprocessor such as the variable selection heuristic and the limit of the number of steps. We performed several experiments in order to set the parameters properly.

The *Unit Propagation preprocessor* is similar to the lower bound computed by current systematic search solvers for Max-SAT. The first unit propagation-based lower bound was introduced in [27] for unweighted Max-SAT: Unit propagation is used to detect disjoint mutually inconsistent subsets of clauses. The number of inconsistent subsets is an *underestimation* of the cost of the solution. In [26, 22, 28] a set of resolution-based rules were introduced to transform a Max-SAT problem instance into an equivalent but simpler one. Finally, the work in [18] showed how to transform inconsistent subsets of clauses detected by unit propagation to equivalent subsets using a limited number of Max-SAT resolution steps. Our Unit Propagation Preprocessor is further enhanced with *Probing* [18]. In [25] probing is used to detect mutually inconsistent subsets of clauses in order to compute an underestimation of the solution. Differently, probing in [18] is used to detect mutually inconsistent subsets of clauses in order to create new unit soft clauses after a resolution process.

The *Clique preprocessor* was introduced in [16] for the Maximum Clique problem and we have extended its use to the *binary unate covering problem.* It is based on the recent Unit Rule [16] and the Star Rule. The Star Rule was introduced in [30] in a general form. It was applied as an underestimation in [1], restricted to clauses of size 2 in [22, 26] and in general in [18]. A variant of the original Clique preprocessor returns a powerful lower bound equal to optimal solution for all HOS [49] and several Scheduling [24] instances. The resulting problem instances have been solved and their optimality has been certified by the first time with a SLS algorithm. Recall that the HOS [49] instances were used in different International Evaluations [4] and previous works [31] but most of them remained unresolved.

9. Conclusions and future work

In this paper, three preprocessors have been presented based on the Max-SAT resolution rule. The first one is based on the well-known variable saturation algorithm [3]. The second one is based on unit propagation and it is widely used in current systematic search algorithms to boost the search [18, 26]. The third one is an extended version of the preprocessor presented in [16] to handle *binary unate covering problems.* We analyzed the performance of stochastic local search algorithms when they are fed with the preprocessed instances.

We observed that the variable saturation preprocessor does not report significant improvements. Besides, best improvement algorithms like Gsat and Irots are less sensitive to inference. Differently, algorithms based on the Walksat architecture like Walksat, Adaptnovely+ and Saps report significant improvements for the Unit Propagation and Clique preprocessors. Furthermore, we observed that some preprocessed instances by the Clique preprocessor can be solved with a local search algorithm to optimality when the weight of the empty clause is equal to the value of the optimal solution.

New research venues are open. First, we plan to investigate how effective could be powering local search solvers with inference not only as a preprocessor but also at each search step. Second, it would be interesting to find a heuristic method for applying the Clique preprocessor so that it returns an empty clause with a weight equal to the optimal solution more frequently as it happens for some problem instances with $Cl+$. The heuristic may be based on the underlying graph defined by the binary hard clauses. It may allow local search solvers to prove the optimality as we have pointed out in this paper. Finally, we would like to extend this research to problems with non-Boolean variables. For example, we can consider the WCSP framework [23] where variables have finite domains. In this context, we can study the effect of *soft local consistency* on existing local search solvers [29].

References

[1] T. Alsinet, F. Manyà, and J. Planes. An efficient solver for weighted Max-SAT. *J. Global Optimization*, **41**(1):61–73, 2008.

[2] Anbulagan, D. Nghia Pham, J. K. Slaney, and A. Sattar. Old resolution meets modern SLS. In *AAAI*, pages 354–359, 2005.

[3] J. Argelich, C. Li, and F. Manyà. A preprocessor for Max-SAT solvers. In *SAT*, pages 15–20, 2008.

[4] J. Argelich, C. Li, F. Manyà, and J. Planes. The first and second Max-SAT evaluations. *JSAT*, **4**(2-4):251–278, 2008.

[5] R. Battiti and M. Protasi. Reactive search, a history-sensitive heuristic for Max-SAT. *ACM Journal of Experimental Algorithmics*, **2**:2, 1997.

[6] J. R. Bitner and E. M. Reingold. Backtrack programming techniques. *Commun. ACM*, **18**(11):651–656, 1975.

[7] M. L. Bonet, J. Levy, and F. Manyà. Resolution for Max-SAT. *Artif. Intell.*, **171**(8-9):606–618, 2007.

[8] S. Cai, K. Su, and Q. Chen. Ewls: A new local search for minimum vertex cover. In *AAAI*, pages 45–50, 2010.

[9] B. Cha and K. Iwama. Adding new clauses for faster local search. In *Proc. of the 13th AAAI*, pages 332–337, Portland, OR, 1996.

[10] O. Coudert. On solving covering problems. In *Design Automation Conference, DAC-96*, pages 197–202, 1996.

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

[12] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA*, pages 485–498, 2002.

[13] H. Fang and W. Ruml. Complete local search for propositional satisfiability. In *AAAI*, pages 161–166, 2004.

[14] A. Guerri and M. Milano. CP-IP techniques for the bid evaluation in combinatorial auctions. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming, CP-03*, pages 863–867, 2003.

[15] F. Heras and J. Larrosa. Intelligent variable orderings and re-orderings in dac-based solvers for WCSP. *Journal of heuristics*, **12**:287–306, 2006.

[16] F. Heras and J. Larrosa. A Max-SAT inference-based pre-processing for max-clique. In *SAT*, pages 139–152, 2008.

[17] F. Heras, J. Larrosa, S. de Givry, and T. Schiex. 2006 and 2007 Max-SAT evaluations: Contributed instances. *JSAT*, **4**(2-4):239–250, 2008.

[18] F. Heras, J. Larrosa, and A. Oliveras. Minimaxsat: An efficient weighted Max-SAT solver. *J. Artif. Intell. Res. (JAIR)*, **31**:1–32, 2008.

[19] H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *AAAI/IAAI*, pages 655–660, 2002.

[20] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In *AAAI/IAAI*, pages 661–666, 1999.

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

[22] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient Max-SAT solving. *Artificial Intelligence*, **172**(2-3):204–233, 2008.

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

[24] K. Leyton-Brown. Cats generator. `http://www.cs.ubc.ca/~kevinlb/CATS/`, 2003.

[25] C. M. Li, F. Manyà, and J. Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *AAAI*, 2006.

[26] C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *J. Artif. Intell. Res. (JAIR)*, **30**:321–359, 2007.

[27] C. M. Li, F. Manyà, and J. Planes. Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In *Proc. of the 11th CP*, Sitges, Spain, 2005.

[28] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Transforming inconsistent subformulas in maxsat lower bound computation. In *CP*, pages 582–587, 2008.

[29] B. Neveu and G. Trombettoni. When local search goes with the winners. In *5th Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR 2003*, 2003.

[30] R. Niedermeier and P. Rossmanith. New upper bounds for maximum satisfiability. *J. Algorithms*, **36**(1):63–88, 2000.

[31] G. Palubeckis. Solving the weighted Max-2-SAT problem with iterated tabu search. *Journal of Information Technology and Control*, **37**:275–284, 2008.

[32] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, USA, 1994.

[33] J. D. Park. Using weighted max-SAT engines to solve MPE. In *Proc. of the 18th AAAI*, pages 682–687, Edmonton, Alberta, Canada, 2002.

[34] M. Pipponzi and F. Somenzi. An iterative algorithm for the binate covering problem. In *EURO-DAC '90: Proceedings of the conference on European design automation*, pages 208–211, 1990.

[35] I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *J. Autom. Reasoning*, **24**(1/2), 2000.

[36] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, pages 542–547, 1999.

[37] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings of the second DIMACS Challenges on Cliques, Coloring and Satisfiability*, 1993.

[38] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence, AAAI-92*, pages 440–446, 1992.

[39] K. Smyth, H. H. Hoos, and Thomas Stützle. Iterated robust tabu search for Max-SAT. In *Canadian Conference on AI*, pages 129–144, 2003.

[40] D. M. Strickland, E. Barnes, and J. S. Sokol. Optimal protein structure alignment using maximum cliques. *Operations Research*, **53**(3):389–402, 2005.

[41] E. D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, **17**(4-5):443–455, 1991.

[42] D. A. D. Tompkins and H. H. Hoos. Ubcsat: An implementation and experimentation environment for SLS algorithms for SAT & Max-SAT. In *SAT*, 2004.

[43] D. A. D. Tompkins and H. H. Hoos. Scaling and probabilistic smoothing: Dynamic local search for unweighted Max-SAT. In *Canadian Conference on AI*, pages 145–159, 2003.

[44] A. van Gelder. Cnfgen formula generator. `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/`, 1993.

[45] M. Vasquez and 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.

[46] H. Xu, R. A. Rutenbar, and K. A. Sakallah. sub-SAT: a formulation for relaxed Boolean satisfiability with applications in routing. *IEEE Trans. on CAD of Integrated Circuits and Systems*, **22**(6):814–820, 2003.

[47] K. Xu. Hidden optimal solution benchmarks. `http://www.nlsde.buaa.edu.cn/~kexu/`, 2005.

[48] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *IJCAI*, pages 337–342, 2005.

[49] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artif. Intell.*, **171**(8-9):514–534, 2007.

[50] M. Yagiura and T. Ibaraki. Efficient 2 and 3-flip neighborhood search algorithms for the Max-SAT: Experimental evaluation. *J. Heuristics*, **7**(5):423–442, 2001.

[51] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computed-Aided design, ICCAD-01*, pages 279–285, 2001.