

On Inconsistent Clause-Subsets for Max-SAT Solving*

Sylvain Darras, Gilles Dequen, Laure Devendeville, and Chu-Min Li

LaRIA, FRE 2733 – Université de Picardie Jules Verne
33, Rue St-Leu, 80039 Amiens Cedex 01, France
{sylvain.darras, gilles.dequen, laure.devendeville,
chu-min.li}@u-picardie.fr

Abstract. Recent research has focused on using the power of look-ahead to speed up the resolution of the Max-SAT problem. Indeed, look-ahead techniques such as Unit Propagation (UP) allow to find conflicts and to quickly reach the upper bound in a Branch-and-Bound algorithm, reducing the search-space of the resolution. In previous works, the Max-SAT solvers *maxsatz9* and *maxsatz14* use unit propagation to compute, at each node of the branch and bound search-tree, disjoint inconsistent subsets of clauses in the current subformula to estimate the minimum number of clauses that cannot be satisfied by any assignment extended from the current node. The same subsets may still be present in the subtrees, that is why we present in this paper a new method to memorize them and then spare their recomputation time. Furthermore, we propose a heuristic so that the memorized subsets of clauses induce an ordering among unit clauses to detect more inconsistent subsets of clauses. We show that this new approach improves *maxsatz9* and *maxsatz14* and suggest that the approach can also be used to improve other state-of-the-art Max-SAT solvers.

Keywords: Max-SAT, Unit Propagation, Inconsistent Subset.

1 Introduction

The Max-SAT (short for Maximum Satisfiability) problem is to find an assignment of logical values to the variables of a propositional formula expressed in terms of a conjunction of clauses (i.e. disjunction of literals) that satisfies the maximum number of clauses. This optimization problem is a generalization of the Satisfiability Decision problem. During the last decade, the interest in studying Max-SAT has grown significantly. These works highlight Max-SAT implications in real problems, as diverse as scheduling [1], routing [2], bioinformatics [3], . . . , motivating considerable progresses concerning efficient Max-SAT solving based on the Branch-and-Bound (BnB) scheme.

The almost common characteristic of the successive progresses for the BnB scheme is to propose new ways of computing the Lower Bound (LB). Given a Max-SAT instance Σ , the BnB algorithm implicitly enumerates the search-space of all possible assignments using a binary search-tree. At every node, BnB compares the Upper Bound (UB), which is the minimum number of conflict clauses in Σ obtained so far for a complete assignment, with LB which is an underestimation of the minimum number of conflict clauses of Σ by any complete assignment obtained by extending the partial assignment at the current node. If $LB \geq UB$, the algorithm prunes the subtree below the

* This work was partially supported by Région Champagne-Ardennes.

current node, since a better solution cannot be obtained from the current node. In previous works, the quality of LB was improved of several manners. Thus, [4–6] define new inference rules based on special cases of resolution that are able to outperform state-of-the-art solvers [7, 8]. In addition, another way to improve LB is to make an intensive use of unit propagation with look-ahead techniques at each node of the BnB algorithm in order to find disjoint inconsistent clause-subsets (named ICS in the following) [9, 10]. These inconsistent cores of clauses allow LB to quickly equal or exceed UB and to decrease the mean-size of the BnB search-tree. However, some characteristics as the mean number of clauses belonging to these cores influence on the quality of LB and consequently on the efficiency of the BnB algorithm. Moreover, the detection of these disjoint inconsistent clause-subsets is not incremental in [9, 10]. In other words, a number of inconsistent clause-subsets detected at a node are detected over and again in the two subtrees of the node.

In this paper, we focus on the study of the characteristics of the inconsistent clause-subsets and their influence on the lower bound quality. After presenting some preliminaries, we study structures and characteristics of inconsistent clause-subsets and propose a heuristic to estimate the usefulness of a given inconsistent subset of clauses in LB calculus. We then describe our heuristic-based approach to improve LB and to make the LB calculus more incremental. This approach is applied to the Max-SAT solvers *maxsatz9* [10] and *maxsatz14* [4], and experimental results on random and Max-Cut instances where some are from the 2007 Max-SAT evaluation¹ show that our approach significantly improves *maxsatz9* and *maxsatz14*.

2 Preliminaries

A *CNF-formula* (Conjunctive Normal Form formula) Σ is a conjunction of clauses where each clause is a disjunction of literals. A literal is the (positively or negatively) signed form of a boolean variable. An *interpretation* of Σ is an assignment over the truth values space $\{True, False\}$ to its variables. It is a partial interpretation if only a subset of the variables of Σ are assigned. A positive (resp. negative) literal is satisfied if the corresponding variable has the value *True* (resp. *False*). A clause is satisfied if at least one of its literals is satisfied. Σ is satisfied according to an interpretation if all its clauses are satisfied. The Max-SAT problem is to find an assignment which minimizes the number of falsified clauses of Σ . This optimization problem is a generalization of the Satisfiability decision problem which is to determine whether a satisfying assignment of all clauses of Σ exists or not.

In this paper, each further reference of *formula* means *CNF-formula*. Moreover, a conjunction of clauses $c_1 \wedge c_2 \wedge \dots \wedge c_m$ is simply represented by a set of clauses $\{c_1, c_2, \dots, c_m\}$. We denote *k-cl* a clause which contains exactly k literals. The 1-*cl* and 2-*cl* clause may be respectively named *unit* and *binary* clause. A *k-SAT* formula is a formula which exclusively consists of *k-cl*. An empty clause contains no literal; it represents a conflict since it cannot be satisfied. When one literal l is assigned to *True*, applying one-literal rule consists in satisfying all clauses containing l (i.e. removing them from the formula) and removing all $\neg l$ from all remaining clauses. The *Unit Propagation* (denoted *UP*) is the iterative process which consists in applying one-

¹ <http://www.maxsat07.udl.es/>

literal rule to each literal appearing in at least one unit clause. This process continues until all unit clauses disappear from Σ or an empty clause is derived.

We focus our work on complete Max-SAT solvers which are based on a BnB algorithm. This class of solvers actually provides the best performances on Max-SAT solving². Given Σ , these solvers implicitly enumerate the whole search-space of the formula by constructing a depth-first search-tree. At every node, they use two important values: UB , the minimum number of conflict clauses in Σ obtained so far for a complete assignment, and LB , an underestimation of the minimum number of conflict clauses of Σ by any complete assignment extending the partial assignment at the current node. If $LB < UB$, the current partial assignment is extended by choosing a decision variable x , which is successively set to *True* and *False* in order to split the current sub-formula into two new simplified ones using the one-literal rule, the process being recursively continued from each of the two new formulas. If $LB \geq UB$ the solvers prune the subtree under the current node, since all complete assignments extending the current partial assignment are worse than the best one found so far. The solution of a BnB solver is the assignment with the minimum UB after enumerating all assignments.

As it allows to prune subtrees, LB is essential in efficient Max-SAT solving. In [11], LB is defined as follows:

$$LB(\Sigma_A) = \#Empty(\Sigma_A) + \sum_{x \in \Sigma_A} \min(\#Unit(x, \Sigma_A), \#Unit(\neg x, \Sigma_A)) \quad (1)$$

where $\#S$ is the number of elements of the set S , Σ_A is the current state of a formula Σ according to a partial assignment A , $Empty(\Sigma_A)$ is the set of empty clauses of Σ_A , and $Unit(lit, \Sigma_A)$ is the set of unit clauses containing the literal lit . Many works have extended this estimation. The equation 1 can be generalized to:

$$LB(\Sigma_A) = \#Empty(\Sigma_A) + \#ICS(\Sigma_A) \quad (2)$$

where $ICS(\Sigma_A)$ is a set of disjoint inconsistent clause-subsets from the formula Σ_A . Subsets of the form $\{x, \neg x\}$, $x \in \Sigma_A$ are included in $ICS(\Sigma_A)$. Thanks to binary clauses using the Directional Arc Consistency notion (DAC) defined in [12] for Max-CSP, [13, 14] have improved the LB computation.

More recently, the equation 2 has led to new rules:

1. The Star rule ([15, 16]) consists in searching inconsistent cores of clauses of the form $\{l_1, \dots, l_k, \neg l_1 \vee \dots \vee \neg l_k\}$.
2. The UP-rule [9] detects inconsistent clause-subsets in Σ_A using unit propagation. The detection can be described as follows. Let Σ_{A1} be a copy of Σ_A . Unit propagation repeatedly applies one-literal rule to simplify Σ_{A1} until there is no more unit clause or an empty clause is derived. If an empty clause is derived, let S be the set of clauses used to derive the empty clause. Then S is an inconsistent subset of clauses, i.e., all clauses in S cannot simultaneously be satisfied by any assignment. For example, let us consider the following set of clauses of a formula Σ_1 :

$$\Sigma_1 = \left\{ \begin{array}{lll} c_1 : x_1 & c_4 : \neg x_2 \vee \neg x_4 & c_7 : \neg x_4 \vee x_6 \\ c_2 : \neg x_1 \vee x_2 & c_5 : \neg x_1 \vee \neg x_3 \vee x_4 & \\ c_3 : \neg x_1 \vee \neg x_2 \vee x_3 & c_6 : \neg x_1 \vee x_5 & \end{array} \right\}$$

² see <http://www.iiia.csic.es/~maxsat06/>

Following the UP process, x_1 has to be *True*, c_2 and c_6 become unit clauses, and so on. Finally, c_5 is empty and the set $S = \{c_1, c_2, c_3, c_4, c_5\}$ is an inconsistent subset of clauses.

The detection of other disjoint inconsistent subformula continues, after excluding the clauses of S , until no more empty clause can be derived. LB is then the number of empty clauses in Σ_A plus the total number of disjoint inconsistent subsets of clauses detected.

3. The failed literals technique [10] (denoted UP_{FL}^*) is an extension of UP -rule. After applying UP -rule to Σ_A to detect disjoint subsets of clauses, it detects additional disjoint inconsistent subsets of clauses by finding failed literals after excluding the inconsistent subsets of clauses already found from Σ_A . This detection can be illustrated as follows. If unit propagation in $\Sigma_A \cup \{x\}$ and unit propagation in $\Sigma_A \cup \{\neg x\}$ both lead to a conflict then clauses in Σ_A implying the two contradictions constitute an inconsistent subset.

The UP_{FL}^* heuristic has been proved much more effective in *maxsatz9* [10] and *maxsatz14* [4] than previous state-of-the-art lower bounds. However, the LB calculus based on UP_{FL}^* in *maxsatz9* and *maxsatz14* is not incremental (in *maxsatz14*, an incremental part of the LB is computed by *inference rules*; we mention them in section 5.2). When LB is still less than UB , the solver should extend the current partial assignment i.e., the solver should branch. Before branching to a new node, all the inconsistent clauses-subset are erased. The new node does not inherit neither the inconsistent clause-subsets detection of its parent, nor the information of these inconsistent clause-subsets such as their size. The UP_{FL}^* function is entirely re-executed at the new node.

The aim of our work is to enable a search-tree node to inherit some inconsistent clause-subsets of its parent, which avoids the entire re-execution of the UP_{FL}^* function and could improve the quality of LB .

3 Improving UP_{FL}^*

The purpose of UP_{FL}^* is to detect as many inconsistent clause-subsets as possible in a formula. Since an inconsistent subset of clauses has to be excluded before detecting other inconsistent subsets of clauses, UP_{FL}^* should detect small enough inconsistent subsets of clauses, leaving more clauses in the formula to facilitate the detection of other inconsistent subsets of clauses. UP_{FL}^* uses a heuristic on the ordering of unit clauses during unit propagation to detect small inconsistent clause-subsets, but when there are several unit clauses in the formula, it selects one in an undetermined ordering to start an unit propagation. However, we believe that the number of inconsistent subsets detected by UP_{FL}^* is strongly correlated to the ordering of initial unit clauses. Moreover, as we noticed above, UP_{FL}^* recomputes the whole inconsistent subset detection in every node of a search-tree, which is time-consuming.

For example, let Σ_2 be the following set of clauses

$$\Sigma_2 = \left\{ \begin{array}{cccc} c_1 : x_1 & c_5 : \neg x_3 \vee x_5 & c_9 : \neg x_6 \vee x_4 & c_{13} : \neg x_8 \vee x_9 \\ c_2 : \neg x_1 \vee x_2 & c_6 : \neg x_4 \vee \neg x_5 & c_{10} : \neg x_6 \vee x_5 & c_{14} : \neg x_8 \vee x_{10} \\ c_3 : \neg x_2 \vee x_3 & c_7 : \neg x_9 \vee \neg x_{10} & c_{11} : \neg x_3 \vee x_7 & \\ c_4 : \neg x_3 \vee x_4 & c_8 : x_6 & c_{12} : \neg x_7 \vee x_8 & \end{array} \right\}$$

1. *unit clause ordering in UP_{FL}^* detection.*

- UP_{FL}^* first calls *UP-rule* detection. Following the empirical UP_{FL}^* techniques of the *maxsatz* solver, c_1 is propagated first, then new unit clauses produced during the unit propagation are stored in a queue and propagated in a first in first out ordering. Thus, the UP-rule first assigns $x_1 = True$ through the clause c_1 . The variables $x_2 = True$, $x_3 = True$, $x_4 = True$, $x_5 = True$, and $x_7 = True$, through c_1 , c_2 , c_3 , c_4 , c_5 , c_{11} respectively. We then get an empty clause c_6 and the inconsistent subset of clauses S is equal to $\{c_1, c_2, c_3, c_4, c_5, c_6\}$. Only clauses $\{c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}\}$ can participate in finding other inconsistent clause-subsets. However, while propagating c_8 , the UP-rule assigns $x_6 = True$, $x_4 = True$, $x_5 = True$ through c_8 , c_9 , c_{10} , respectively, no more conflict is found.
- Consider now another ordering of the unit clauses in Σ_2 that propagates first c_8 . The following assignments are then performed: $x_6 = True$, $x_4 = True$, and $x_5 = True$ through c_8 , c_9 , and c_{10} , respectively. We encounter an empty clause c_6 . The associated inconsistent subset of clauses is $S' = \{c_6, c_8, c_9, c_{10}\}$. The candidates clauses belonging to other inconsistent clause-subsets are then $\{c_1, c_2, c_3, c_4, c_5, c_7, c_{11}, c_{12}, c_{13}, c_{14}\}$. We then propagate c_1 , $x_1 = True$, $x_2 = True$, $x_3 = True$, $x_4 = True$, $x_5 = True$, $x_7 = True$, $x_8 = True$, $x_9 = True$, and $x_{10} = True$ through c_1 , c_2 , c_3 , c_4 , c_5 , c_{11} , c_{12} , c_{13} , and c_{14} , respectively. As c_7 is now empty, a second and disjoint inconsistent clause-subset $S'' = \{c_1, c_2, c_3, c_7, c_{11}, c_{12}, c_{13}, c_{14}\}$ is obtained.

2. *Re-computation of inconsistent clause-subsets.*

Let us consider now the same subformula Σ_2 and assuming that the next decision variable chosen by the solver does not belong to $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$. Σ_2 is unchanged. UP_{FL}^* probably finds the same conflicts, so the re-computation could be avoided if the inconsistent clause-subsets of the parent node were memorized.

Within a practical framework, we propose a method to solve totally or partially the UP_{FL}^* weaknesses mentioned above. The idea is to memorize the small inconsistent subsets of clauses computed at a given node and maintain them for all the corresponding subtrees. This allows to avoid costly re-detection of inconsistent subsets and induce a natural ordering UP_{FL}^* detection. Indeed, storing (small) inconsistent subsets of clauses is equivalent to grant a privilege to the variables which minimize the number of clauses to detect a conflict. We could just store a sorted list of variables, which would give us the same "good" conflictual clause subsets at each node than if we had memorized these sets: it draws a variable ordering. Thus, the goal of our approach is double:

- Inconsistent subsets of clauses stored at a node of the search-tree will not be re-computed at the child-nodes. These subsets are directly added to LB . Once a small inconsistent subset is found, the solver does not spend time to detect it again.
- Since we will empirically determine "interesting" inconsistent clause-subsets, UP_{FL}^* begins its computation in the child-nodes with the stored interesting subsets. In addition to the advantage that these interesting inconsistent clause-subsets are given without specific detection, they induce a natural ordering of unit clauses for detecting small inconsistent subsets of clauses so as to improve LB .

For example, let Σ_3 be the following set of clauses

$$\Sigma_3 = \left\{ \begin{array}{llll} c_1 : x_1 & c_5 : \neg x_3 \vee x_5 & c_9 : \neg x_6 \vee x_4 & c_{13} : \neg x_8 \vee x_9 \\ c_2 : \neg x_1 \vee x_2 & c_6 : \neg x_4 \vee \neg x_5 & c_{10} : \neg x_6 \vee x_5 & c_{14} : \neg x_8 \vee x_{10} \\ c_3 : \neg x_2 \vee x_3 \vee x_{11} & c_7 : \neg x_9 \vee \neg x_{10} & c_{11} : \neg x_3 \vee x_7 & \\ c_4 : \neg x_3 \vee x_4 & c_8 : x_6 & c_{12} : \neg x_7 \vee x_8 & \end{array} \right\}$$

Propagating c_1 , UP_{FL}^* does not find any empty clause. Propagating c_8 , UP_{FL}^* finds an inconsistent subset $S' = \{c_6, c_8, c_9, c_{10}\}$ as for Σ_2 in the previous example, which we store. After branching next on x_{11} and assigning *False* to x_{11} , Σ_3 becomes Σ_2 . Without the storage of S' , UP_{FL}^* propagates c_1 and finds a unique inconsistent clause-subsets $S = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ as in the previous example, since after the clauses in S are removed from Σ_2 , no more conflict is found using unit propagation.

However, with the storage of the inconsistent subset $S' = \{c_6, c_8, c_9, c_{10}\}$, after branching next on x_{11} and assigning *False* to x_{11} , S' is first counted in LB and its clauses are excluded in the conflict detection. UP_{FL}^* naturally detects the second inconsistent subset $S'' = \{c_1, c_2, c_3, c_7, c_{11}, c_{12}, c_{13}, c_{14}\}$.

The above example shows that by memorizing small inconsistent subsets of clauses and "forget" large ones, we are able to detect more inconsistent subsets of clauses.

4 Clause sets storage

We have seen in the previous section that the storage of inconsistent subsets of clauses and their reuse are helpful. We should now answer two questions: (i) should we memorize all inconsistent subsets of clauses for child-nodes? (ii) if not, what are inconsistent subsets of clauses that should be stored? For this purpose, we should analyze the impact of branching on a variable.

Let us consider an inconsistent subset $S_k = \{c_{k_1}, c_{k_2}, \dots, c_{k_n}\}$ found at a node of the search-tree by UP_{FL}^* . Note that S_k is a minimal inconsistent subset of clauses in the sense that it becomes consistent if any clause is removed from it, meaning that any literal in S_k should also have its negated literal in S_k . Let x be the next branching variable. We can distinguish three cases:

1. Variable x appears in clauses of S_k .

In this case, the literals x and $\neg x$ belong to S_k . S_k has no meaning anymore after branching because it contains satisfied clauses, and S_k remains inconsistent but is not probably minimal anymore. We should then find the minimal inconsistent subset of S_k and allow other clauses of S_k to be used in the detection of other conflicts.

To illustrate this case, let us consider the following inconsistent clause subset:

$$S_k = \left\{ \begin{array}{lll} c_1 : \neg x \vee y & c_3 : \neg y \vee \neg z & c_5 : \neg z \vee t \\ c_2 : \neg y \vee z & c_4 : x \vee z & c_6 : \neg z \vee \neg t \end{array} \right\}$$

On the one hand, when x is set to *True*, the clause c_4 is then satisfied. The subset of clauses $\{y, \neg y \vee z, \neg y \vee \neg z\}$ from c_1 , c_2 and c_3 is inconsistent. This allows the set of clauses $\mathcal{R} = \{\neg z \vee t, \neg z \vee \neg t\}$ from c_5 and c_6 to take part of other conflictual sets. On the other hand, when x is set to *False*, the subset of clauses

$\{z, \neg z \vee t, \neg z \vee \neg t\}$ from c_4, c_5 and c_6 is inconsistent and $\mathcal{R} = \{\neg y \vee z, \neg y \vee \neg z\}$ from c_2 and c_3 can be used to detect other conflicts.

2. Variable x does not appear in clauses of the S_k , but some of the clauses shortened by the assignment of x form a smaller inconsistent subset S'_k using some clauses of S_k .

S'_k and S_k are inconsistent but not disjoint. Only one of S'_k and S_k can contribute to increase LB . It would be more useful to keep S'_k instead of S_k .

For example, consider the inconsistent clause-subset S_k such as:

$$S_k = \left\{ \begin{array}{lll} c_1 : \neg w \vee y & c_3 : \neg y \vee \neg z & c_5 : \neg z \vee t \\ c_2 : \neg y \vee z & c_4 : w \vee z & c_6 : \neg z \vee \neg t \end{array} \right\}$$

and the two clauses $c_7 : \neg x \vee z \vee t$ and $c_8 : \neg x \vee z \vee \neg t$. After branching to x and $x=True$, c_7 and c_8 are reduced to $z \vee t$ and $z \vee \neg t$, respectively. Thus, one can exhibit S'_k which consists of:

$$S'_k = \left\{ \begin{array}{ll} c_5 : \neg z \vee t & c_7 : z \vee t \\ c_6 : \neg z \vee \neg t & c_8 : z \vee \neg t \end{array} \right\}$$

In order to maximize the possibility of finding other inconsistent clause-subsets, a solver should forget S_k in order to detect S'_k , allowing clauses in $S_k \setminus S'_k$ to participate in further conflict detection.

3. Variable x does not appear in clauses of S_k and there is no new smaller inconsistent subset using clauses of S_k .

In this third case, it is generally useful to memorize S_k , since otherwise UP_{FL}^* would probably re-detect it at the child-nodes. Memorizing it avoids this re-detection.

Given a formula Σ , the first aim of our work is to induce a natural ordering for unit clauses of Σ so that as many inconsistent subsets of clauses as possible are detected. The second aim is to avoid the repeated detections of the same inconsistent subsets of clauses at different nodes of a search-tree to make the LB computation more incremental. For these two aims, we should keep all inconsistent subsets of clauses in case 3 and avoid keeping any inconsistent subset of clauses in case 1 and case 2 when branching.

Figure 1 shows the branch and bound algorithm for Max-SAT with storage of inconsistent subsets of clauses. The first call to the function is $\text{max-sat}(\Sigma, \#clauses, \emptyset)$. Given a formula Σ , the algorithm begins by removing all clauses in each inconsistent subset stored in $ICSS$ (lines 4-6). Then it computes the set C of further inconsistent subsets of clauses using UP_{FL}^* (line 7). If LB reaches UB the solver backtracks, otherwise all removed clauses are reinserted (lines 13-15). The heuristic $decideStorage(S)$ decides if the new inconsistent subset of clauses S should be stored (lines 17-19). This heuristic is essentially based on the probability that S is in case 2. Finally, in lines 21-25, all inconsistent subsets containing the next branching variable x are removed from $ICSS$ before branching so that any inconsistent subset of clauses in case 1 is not stored anymore.

Since all inconsistent subsets of clauses containing the next branching variable x (case 1) are excluded, the heuristic $decideStorage(S)$ only decides if S would probably lead to the case 2. If the heuristic performs well, most inconsistent subsets of clauses stored in $ICSS$ should be in case 3 and consequently an effective lower bound LB can be obtained. So the heuristic $decideStorage(S)$ is essential in our approach.

max-sat($\Sigma, UB, ICSS$)

Require: A CNF formula Σ , an upper bound UB , and a set of inconsistent subsets of clauses $ICSS$

Ensure: The minimal number of unsatisfied clauses of Σ

```

1: if  $\Sigma = \emptyset$  or  $\Sigma$  only contains empty clauses then
2:   return  $\#Empty(\Sigma)$ ;
3: end if
4: for all inconsistent subset  $S$  in  $ICSS$  do
5:   remove clauses of  $S$  from  $\Sigma$ 
6: end for
7:  $C \leftarrow UP_{FL}^*(\Sigma)$ ;
8:  $LB \leftarrow \#Empty(\Sigma) + \#ICSS + \#C$ 
9: if  $LB \geq UB$  then
10:  return  $\infty$ ;
11: end if
12:  $x \leftarrow selectVariable(\Sigma)$ 
13: for all inconsistent subset  $S$  in  $ICSS$  do
14:  reinsert clauses of  $S$  into  $\Sigma$ 
15: end for
16: for all inconsistent subset  $S$  in  $C$  do
17:  if  $decideStorage(S)=True$  then
18:     $ICSS \leftarrow ICSS \cup \{S\}$ 
19:  end if
20: end for
21: for all inconsistent subset  $S$  in  $ICSS$  do
22:  if  $S$  contains  $x$  (case 1) then
23:    remove  $S$  from  $ICSS$ 
24:  end if
25: end for
26:  $UB \leftarrow \min(UB, max-sat(\Sigma_{\bar{x}}, UB, ICSS))$ 
27: return  $\min(UB, max-sat(\Sigma_x, UB, ICSS))$ 

```

Fig. 1. A branch and bound algorithm for Max-SAT with storage of inconsistent subsets of clauses

It is obviously not conceivable to build a function that could definitely decide if a clause subset would never hide a smaller set since it would need to consider every node of the subtrees rooted at the current node. So we rather use a heuristic to define $decideStorage(S)$.

We conjecture that the more clauses an inconsistent clause-subset S contains, the higher is the probability that some of its clauses are involved in a smaller new inconsistent subset of clauses after branching. In other words, a larger inconsistent clause-subset S has higher probability to be in case 2 and a smaller inconsistent clause-subset S has higher probability to be in case 3. This conjecture suggests that the size of S is an effective characteristic to be used in $decideStorage(S)$ to decide if S is in case 3 and should be stored. Our empirical results in the next section seem to confirm this conjecture.

5 Experimental results

As indicated in Algorithm 1, our approach can be implemented in all branch-and-bound methods dedicated to complete Max-SAT solving of which the LB computation is based on the detection of disjoint inconsistent subsets of clauses. We have chosen to

graft it into the *maxsatz* solver [10] which is the best performing Max-SAT solver in the Max-SAT evaluation 2006³. Nevertheless, in order to validate the strength of our approach, we have tested it on the two last versions of *maxsatz*: version 9 with "Failed Literals" [10] (denoted *maxsatz9*) and version 14 with "Failed Literals" and "Inference Rules" [4] (denoted *maxsatz14*). These two solvers extended by our approach are respectively named *maxsatz9_{icss}* and *maxsatz14_{icss}* in the following, where *icss* denotes Inconsistent Clause Subset Storage. Moreover, we compare *maxsatz9_{icss}* and *maxsatz14_{icss}* with *Toolbar_{v3}* [17, 18] and *MaxSolver* [19], two other state-of-the-art Max-SAT solvers. The experimentations have been done on a 2.6 GHz Bi-Opteron with 2 GB of RAM under Linux OS (kernel 2.6).

5.1 Size of the inconsistent clause-subsets

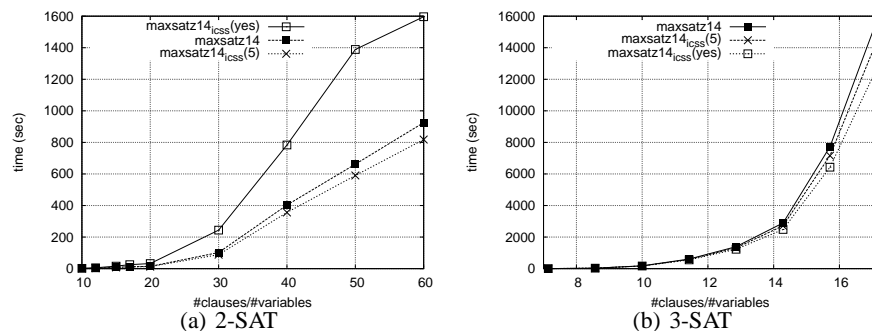


Fig. 2. mean computation time of the *maxsatz14* solver with or without inconsistent subsets storage for randomly generated 2-SAT formulae and 3-SAT formulae with 80 variables with the ratio $\frac{\#clauses}{\#vars}$ from 10 to 80 (2-SAT) and from 7 to 17 (3-SAT). The *decideStorage* heuristic chooses to store all inconsistent subsets (*maxsatz14_{icss}(yes)*) or the inconsistent subsets of at most five clauses (*maxsatz14_{icss}(5)*) with at most two unit clauses, respectively

We use a heuristic based on the size of an inconsistent subset S of clauses to indicate whether S is in case 3 or not. We empirically determine that S should probably be in case 3 if it contains at most five clauses of which at most two unit clauses. Figure 2 shows the influence of the storage of inconsistent clause-subsets according to their size. It compares the mean run-time of the *maxsatz14* solver on sets of 2-SAT and 3-SAT formulae with 80 variables when the $\frac{\#clauses}{\#vars}$ ratio increases, where the *decideStorage* predicate in algorithm 1 chooses to store inconsistent subsets with at most five clauses (*maxsatz9/14_{icss}(5)*), of which at most two unit clauses, and with all inconsistent subsets (*maxsatz9/14_{icss}(yes)*), with at most two unit clauses, respectively. The reason of limiting the number of unit clauses in a stored inconsistent subset is that an unit clause is likelier to be involved in other inconsistent subsets detected by unit propagation, so that inconsistent subsets containing more than two unit clauses should not be stored, since these unit clauses can be used to possibly detect more inconsistent subsets after branching. We have developed a *maxsatz9/14_{icss}(yes)* without limit on the number of unit clauses which gives the worst results on each instance. Note that both

³ see <http://www.iiaa.csic.es/~maxsat06/>

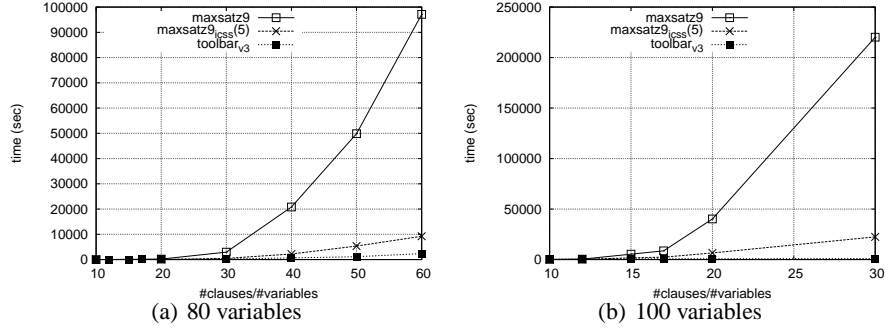


Fig. 3. mean computation time of *maxsatz9* and *maxsatz9_{ics}(5)* on sets of randomly generated 2-SAT formulae of 80 and 100 variables according to the ratio $\frac{\#clauses}{\#vars}$ from 10 to 60/30

maxsatz14_{ics}(5) and *maxsatz14_{ics}(yes)* are based on *maxsatz14*, the currently fastest solver for Max-2-SAT and Max-3-SAT in our knowledge.

Considering the 2-SAT curve with high $\frac{\#clauses}{\#vars}$ ratios in Figure 2(a), the mean time is more than twice faster when the *decideStorage* predicate chooses to store subsets with a low number of clauses (see *maxsatz14_{ics}(5)* curve) than for an unlimited storage (see *maxsatz14_{ics}(yes)* curve). Moreover, at this point of the curve, the search-tree size of *maxsatz14_{ics}(yes)* is almost three times bigger than the search-tree size of *maxsatz14_{ics}(5)* (see table 4). Note that *maxsatz14_{ics}(yes)* is slower than *maxsatz14* in which no inconsistent subset is stored, but *maxsatz14_{ics}(5)* is faster than *maxsatz14*.

For 3-SAT formulae, it is less easy to find an inconsistent subset of clauses, and it is less probable that a clause in an inconsistent subset to be involved in a smaller subset of clauses. So we observe in Figure 2(b), that it is better to store all detected inconsistent subsets of clauses and *maxsatz14_{ics}(yes)* is the best option. Nevertheless, *maxsatz14_{ics}(5)* is not far from *maxsatz14_{ics}(yes)* and is faster than *maxsatz14*.

We finally note in Figure 2 that when the $\frac{\#clauses}{\#vars}$ ratio of the formula increases, the gain with the inconsistent subsets storage becomes more significant in terms of computation time.

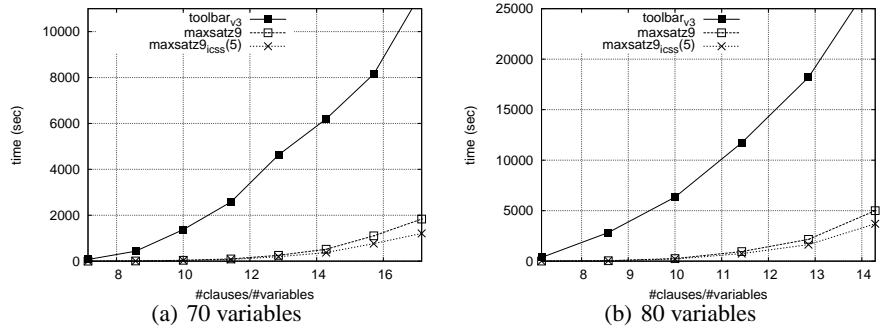


Fig. 4. mean computation time of *maxsatz9*, *maxsatz9_{ics}(5)* and *Toolbar_{v3}* on sets of randomly generated 3-SAT formulae which consists of 70 and 80 variables according to the ratio $\frac{\#clauses}{\#vars}$ from 7 to 17/15

5.2 Comparative results

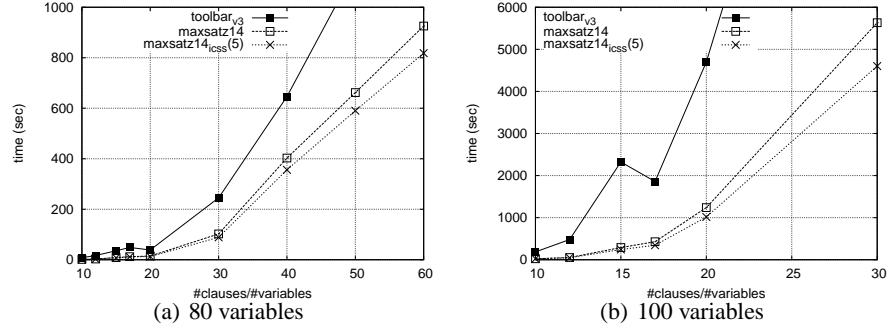


Fig. 5. mean computation time *maxsatz14*, *maxsatz14_{ics(5)}* and *Toolbar_{v3}* on sets of randomly generated 2-SAT formulae of 80 and 100 variables according to the ratio $\frac{\#clauses}{\#vars}$ from 10 to 60/30

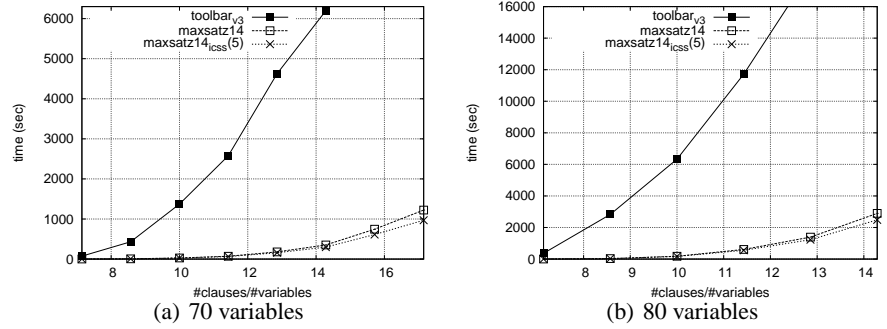


Fig. 6. mean computation time of *maxsatz14*, *maxsatz14_{ics(5)}* and *Toolbar_{v3}* on sets of randomly generated 3-SAT formulae which consists of 70 and 80 variables according to the ratio $\frac{\#clauses}{\#vars}$ from 7 to 17/15

Figure 3 and Figure 4 compare *maxsatz9_{ics(5)}*, *maxsatz9*, and *Toolbar_{v3}* on Max-2-SAT and Max-3-SAT problems. It is clear that with the inconsistent clause-subsets storage, *maxsatz9_{ics(5)}* is significantly faster than *maxsatz9*. Using powerful inference rules, *Toolbar_{v3}* is faster than *maxsatz9_{ics(5)}* and *maxsatz9* for Max-2-SAT, but both *maxsatz9_{ics(5)}* and *maxsatz9* are faster than *Toolbar_{v3}* for Max-3-SAT.

Figure 5 and Figure 6 compare *maxsatz14*, *maxsatz14_{ics(5)}* and *Toolbar_{v3}* on Max-2-SAT and Max-3-SAT problems. Both *maxsatz14*, *maxsatz14_{ics(5)}* are substantially faster than *Toolbar_{v3}* for Max-2-SAT and Max-3-SAT problems and *maxsatz14_{ics(5)}* is significantly faster than *maxsatz14*.

Table 2 and Table 1 show the average computation time improvement (in %) of *maxsatz9_{ics(5)}* and *maxsatz14_{ics(5)}* compared with *maxsatz9* and *maxsatz14*, respectively, for Max-2-SAT and Max-3-SAT problems. It is clear that the improvement becomes more important when the $\frac{\#clauses}{\#vars}$ increases.

Table 4 and Table 3 compare the size of search-trees of *maxsatz9*, *maxsatz14*, *maxsatz9_{ics(5)}*, *maxsatz14_{ics(5)}*, and *Toolbar_{v3}* for Max-2-SAT and Max-3-SAT

<i>solver</i> / $\frac{\#clauses}{\#vars}$	7.14	8.57	10.00	11.43	12.85	14.28	15.71	17.15	18.56	20.00	21.43	22.85
60v/ <i>maxsatz9</i>	11	14	18	22	25	29	32	37	40	43	46	49
70v/ <i>maxsatz9</i>	11	15	18	20	24	27	30	34	38	42	46	
80v/ <i>maxsatz9</i>	12	15	18	20	24	25	29	33				
60v/ <i>maxsatz14</i>	5	5	9	9	13	16	18	21	24	25	27	29
70v/ <i>maxsatz14</i>	3	5	6	9	12	15	17	20	22	24	26	
80v/ <i>maxsatz14</i>	3	5	7	8	11	14	16	19				

Table 1. average computation time improvement (in %) provided by the *icss*(5) approach when it is grafted to the two best versions of *maxsatz* on sets of randomly generated 3-SAT formulae which consists of 60, 70 and 80 variables and for ratio $\frac{\#clauses}{\#vars}$ from 7.14 to 22.85

<i>solver</i> / $\frac{\#clauses}{\#vars}$	10	12	15	17	20	30	40	50	60	70	80
60v/ <i>maxsatz9</i>	34	39	54	59	65	76	80	81	84	82	83
70v/ <i>maxsatz9</i>	36	50	55	57	69	83	86	88	88	87	
80v/ <i>maxsatz9</i>	42	52	57	64	72	84	89	89	90		
100v/ <i>maxsatz9</i>	47	50	68	72	83						
60v/ <i>maxsatz14</i>	2	1	5	3	7	5	6	7	7	6	8
70v/ <i>maxsatz14</i>	-6	2	9	8	6	10	9	9	9	10	
80v/ <i>maxsatz14</i>	13	8	10	7	12	13	11	10	11		
100v/ <i>maxsatz14</i>	0	11	16	19	17	18					

Table 2. average computation time improvement (in %) provided by the *icss*(5) approach when it is grafted to the two best versions of *maxsatz* on sets of randomly generated 2-SAT formulae which consists of 60 to 100 variables and for ratio $\frac{\#clauses}{\#vars}$ from 10 to 80

problems. The inconsistent clause subsets storage allows *maxsatz9_{icss}*(5) and *maxsatz14_{icss}*(5) to compute better lower bounds and to have smaller search-trees.

<i>Algorithms</i> / $\frac{\#clauses}{\#vars}$	8.57	10.00	12.85	14.28	17.15	20.00	21.43	22.85
60v/ <i>maxsatz9</i>	1.21	3.22	14.61	27.44	59.65	130.18	180.79	262.86
60v/ <i>maxsatz9_{icss}</i> (5)	1.18	3.13	14.13	26.12	53.80	109.98	148.24	209.32
60v/ <i>Toolbar_{v3}</i>	115.06	258.54	878.23	1,279.87	2,465.58	4,034.10	4,065.77	4,439.90
60v/ <i>maxsatz14</i>	1.20	2.54	9.88	17.32	33.03	64.01	84.47	114.44
60v/ <i>maxsatz14_{icss}</i> (5)	1.19	2.52	9.83	17.19	32.70	63.27	82.96	112.02
70v/ <i>maxsatz9</i>	4.93	19.61	94.64	166.50	471.22	1,397.55	2,188.83	
70v/ <i>maxsatz9_{icss}</i> (5)	4.75	19.01	92.35	161.23	436.11	1,199.83	1,796.31	
70v/ <i>Toolbar_{v3}</i>	987.95	2,615.78	4,228.95	5,229.80	3,941.12	3,941.12	3,941.12	
70v/ <i>maxsatz14</i>	4.80	14.84	63.75	104.27	268.73	694.23	1,012.11	
70v/ <i>maxsatz14_{icss}</i> (5)	4.78	14.83	63.31	103.21	264.87	681.41	994.00	
80v/ <i>maxsatz9</i>	22.57	113.01	644.61	1,303.28	5,388.91			
80v/ <i>maxsatz9_{icss}</i> (5)	21.64	109.11	632.50	1,278.88	5,095.61			
80v/ <i>Toolbar_{v3}</i>	5,629.36	7,927.10	4,530.50	4,530.50				
80v/ <i>maxsatz14</i>	19.67	81.28	409.06	716.40	2,965.09			
80v/ <i>maxsatz14_{icss}</i> (5)	19.53	80.80	406.62	710.32	2,921.41			

Table 3. Average search-tree size (in 10^4 nodes) of *Toolbar_{v3}*, *maxsatz9*, *maxsatz9_{icss}*(5), *maxsatz14* and *maxsatz14_{icss}*(5) on sets of randomly generated 3-SAT formulae of 60, 70 and 80 variables with ratio $\frac{\#clauses}{\#vars}$ from 8.57 to 22.85

The comparison of *maxsatz14* and *maxsatz14_{icss}*(5) deserves more discussions. *maxsatz14* is *maxsatz9* augmented with 4 inference rules [4]. An inference rule in *maxsatz14* transforms a special inconsistent subset of binary and unit clauses into an empty clause and a number of other clauses, so that the conflict represented by the empty clause does not need to be re-detected in the subtree and the new clauses can be used to detect other conflicts. For example, Rule 3 in *maxsatz14* transforms the set of clauses $\{x_1, x_2, \neg x_1 \vee \neg x_2\}$ into an empty clause and a binary clause $x_1 \vee x_2$. The empty clause contributes to increase the lower bound by one, and the binary clause $x_1 \vee x_2$ can be used to detect other conflicts.

Algorithms/ $\frac{\#clauses}{\#vars}$	10	12	15	17	20	30	40	50	60
60v/ <i>maxsatz9</i>	0.1	0.2	0.6	1.1	1.3	4.7	8.3	13.8	19.0
60v/ <i>maxsatz9</i> _{icss} (5)	0.1	0.1	0.4	0.7	0.8	2.2	3.6	5.5	7.0
60v/ <i>Toolbar</i> _{v3}	0.8	1.1	1.8	3.2	3.0	5.6	8.1	12.9	11.8
60v/ <i>maxsatz14</i>	0.03	0.04	0.09	0.15	0.14	0.27	0.44	0.66	0.71
60v/ <i>maxsatz14</i> _{icss} (5)	0.03	0.04	0.09	0.1	0.1	0.3	0.4	0.6	0.7
70v/ <i>maxsatz9</i>	0.3	0.5	2.4	4.1	10.6	54.1	131.3	282.6	434.3
70v/ <i>maxsatz9</i> _{icss} (5)	0.2	0.4	0.1	0.3	5.6	18.3	41.7	75.1	121.2
70v/ <i>Toolbar</i> _{v3}	2.1	2.6	6.3	12.4	15.6	48.3	46.3	88.3	109.1
70v/ <i>maxsatz14</i>	0.07	0.09	0.3	0.4	0.7	1.9	2.7	4.9	7.1
70v/ <i>maxsatz14</i> _{icss} (5)	0.08	0.1	0.3	0.4	0.8	1.8	2.6	4.7	6.8
80v/ <i>maxsatz9</i>	1.3	5.1	12.3	21.2	25.6	238.9	1,238.2	2,161.4	3,317.6
80v/ <i>maxsatz9</i> _{icss} (5)	1.1	4.0	7.9	11.4	11.2	71.5	283.4	523.1	734.6
80v/ <i>Toolbar</i> _{v3}	14.3	25.7	43.5	51.31	35.19	166.39	345.93	578.73	322.58
80v/ <i>maxsatz14</i>	0.3	0.5	1.2	1.5	1.5	6.3	19.2	24.9	27.8
80v/ <i>maxsatz14</i> _{icss} (5)	0.2	0.5	1.2	1.5	1.4	6.1	18.4	23.7	25.7
100v/ <i>maxsatz9</i>	33.6	69.4	768.4	1,078.4	4,364.8	12,348.8			
100v/ <i>maxsatz9</i> _{icss} (5)	26.0	48.2	324.9	404.6	1,040.2	2,349.1			
100v/ <i>Toolbar</i> _{v3}	236.8	528.8	2,235.3	1,489.5	3,307.4	8,513.2			
100v/ <i>maxsatz14</i>	3.8	7.2	32.1	40.8	95.6	258.3			
100v/ <i>maxsatz14</i> _{icss} (5)	4.1	7.0	29.7	36.9	88.8	233.8			

Table 4. Average search-tree size (in 10^4 nodes) of *Toolbar*_{v3}, *maxsatz9*, *maxsatz9*_{icss}(5), *maxsatz14* and *maxsatz14*_{icss}(5) on sets of randomly generated 2-SAT formulae of 60 to 100 variables with ratio $\frac{\#clauses}{\#vars}$ from 10 to 60

So the empty clause made explicit using an inference rule in *maxsatz14* makes the lower bound computation in different search-tree nodes more incremental, since the conflict does not need to be re-detected in the subtree. The new added clauses using the inference rule allows to improve the lower bound.

The inference rules implemented in *Toolbar*_{v3} are independently designed for weighted Max-SAT, and are similar to those implemented in *maxsatz14* when applied to (unweighted) Max-SAT. Note that all these inference rules are limited to unit and binary clauses.

Unfortunately, the inference rules implemented in *maxsatz14* and *Toolbar*_{v3} are for special subsets of clauses and cannot be applied to transform all inconsistent subsets of clauses. For example, no inference rule is applied in *maxsatz14* and *Toolbar*_{v3} to the inconsistent clause set $\{x_1, x_2, x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3\}$, which has to be re-detected in every node of a search-tree. This set is indeed equivalent to the set $\{\emptyset, x_1 \vee x_2, x_1 \vee x_3, \neg x_1 \vee x_2 \vee x_3\}$, which is time-consuming to detect and to treat. *maxsatz14* is limited to detect some special patterns which are shown practically efficient as inference rules.

Our inconsistent clause subsets storage approach allows to complete the inference rules. In other words, when no inference rule is applicable, the inconsistent clause subset can be stored to make the lower bound computation incremental and to improve the lower bound. The experimental results show the effectiveness of our approach.

We have also tested *MaxSolver* [19], which is too slow to be showed in Figures 3, 4, 5 and 6, and in Tables 2, 1, 4, and 3.

Table 5 compares *MaxSolver*, *maxsatz9*, *maxsatz14*, *maxsatz9*_{icss}(5), *maxsatz14*_{icss}(5), and *Toolbar*_{v3} on max-cut instances submitted to the Max-SAT evaluation 2006. Observe that *maxsatz9*_{icss}(5) is significantly faster than *maxsatz9* and *maxsatz14*_{icss}(5) is also generally faster than *Toolbar*_{v3} and *maxsatz14* as the inconsistent clause-subsets storage complete inference rules in *maxsatz14*.

Benchmarks	<i>maxsatz9</i>	<i>maxsatz9_{icss}(5)</i>	<i>Toolbar_{v3}</i>	<i>maxsolver</i>	<i>maxsatz14</i>	<i>maxsatz14_{icss}(5)</i>
<i>brock200</i>	98.43	38.40	40.33	18,902	7.58	6.58
<i>brock400</i>	350.22	141.00	118.88	> 25,200	27.22	24.49
<i>brock800</i>	53.47	19.37	16.74	18,917	3.21	2.90
<i>c - fat200</i>	1.09	0.48	0.33	2.46	0.07	0.08
<i>c - fat500</i>	10,743.42	10,658.90	12,737	12,615.07	7,313.89	7,353.20
<i>hamming6</i>	12.605	12.601	12,600	> 25,200	12,600.27	12,600.22
<i>hamming8</i>	12,603.77	12,601.53	12,600	12,609.13	12,600.22	12,600.20
<i>hamming10</i>	15,013.98	14,009.33	13,337.14	> 25,200	12,858.69	12,803.17
<i>johnson16</i>	7.40	3.40	1.00	> 25,200	0.39	0.30
<i>johnson32</i>	2,535.20	1,066.03	311.27	> 25,200	127.13	111.03
<i>keller</i>	116.13	45.21	13.02	> 25,200	4.47	3.39
<i>maxcut60420</i>	11.68	2.99	5.88	122.90	0.90	0.88
<i>maxcut60500</i>	68.76	15.86	27.01	523.37	3.00	2.94
<i>maxcut60560</i>	210.64	47.47	67.27	> 25,200	6.40	6.03
<i>maxcut60600</i>	365.98	81.89	96.24	> 25,200	9.20	8.58
<i>phat300</i>	164.27	63.14	81.92	8,400.58	10.82	8.60
<i>phat500</i>	356.30	142.48	165.31	8,404.01	33.06	31.07
<i>phat700</i>	123.76	49.01	54.66	8,400.79	14.60	12.13
<i>phat1000</i>	51.28	20.09	21.80	8,401.15	3.56	2.89
<i>san200₀</i>	4,033.84	1,878.20	1,345.57	> 25,200	451.01	404.92
<i>san400₀</i>	1,250.98	561.10	434.97	20,161.99	127.99	113.26
<i>san1000</i>	4.03	1.62	1.25	15.28	0.23	0.22
<i>sanr200</i>	1,354.37	617.36	516.93	> 25,200	134.10	116.50
<i>sanr400</i>	42.63	15.28	13.17	12,606.03	2.49	1.80
<i>t4pm3 - 6666.spn</i>	0.80	0.03	10.18	0.45	0.05	0.04
<i>t5pm3 - 7777.spn</i>	209.82	51.96	> 25,200	15,336.68	116.72	65.70
<i>ramk3n10.ra0</i>	262.230	261.850	594.46	> 25,200	293.430	294.530
<i>ramk3n9.ra0</i>	0.120	0.120	0.27	> 25,200	0.140	0.140
<i>spinglass4₂.pm3</i>	0.160	0.070	17.81	> 25,200	0.160	0.120
<i>spinglass5₁₀.pm3</i>	190.310	54.200	> 25,200	> 25,200	101.290	60.910
<i>spinglass5₁.pm3</i>	394.140	108.020	> 25,200	> 25,200	229.330	137.820
<i>spinglass5₃.pm3</i>	309.470	87.590	> 25,200	> 25,200	160.510	105.610
<i>spinglass5₅.pm3</i>	105.100	25.300	> 25,200	> 25,200	45.310	25.530
<i>spinglass5₆.pm3</i>	578.410	154.120	> 25,200	> 25,200	343.910	206.920
<i>spinglass5₇.pm3</i>	272.320	61.110	> 25,200	> 25,200	117.880	65.090
<i>spinglass5₉.pm3</i>	162.930	47.600	> 25,200	> 25,200	102.170	62.310

Table 5. mean computation time (in sec.) of *maxsatz9*, *maxsatz9_{icss}(5)*, *maxsatz14*, *maxsatz14_{icss}(5)*, *Toolbar_{v3}* and *MaxSolver* on some families of Max-Cut and Max-SAT evaluation benchmarks

6 Conclusion

We study the detection of disjoint inconsistent subsets of clauses in *maxsatz9* and *maxsatz14* and find that the detection could be more incremental and that the ordering of initial unit clauses in a CNF formula is important for the lower bound computation. We then propose an approach to store some inconsistent clause subsets at a node for its subtrees. The selection of a clause subset S to be stored is determined by a heuristic based on the size of the clause subset, estimating the probability that the clauses of S are used to detect smaller inconsistent subsets than S in the subtrees. The experimental results show that our approach improves *maxsatz9* and *maxsatz14* in terms of runtime and search-tree size, by making the lower bound more incremental and by inducing a natural ordering of initial unit clauses in a CNF formula. Our approach completes the inference rules in *maxsatz14* and could be also used in any Max-SAT solver which computes the lower bound by detecting disjoint inconsistent subsets of clauses.

As future work we also plan to apply the incremental lower bound to Max-SAT solvers that deal with the many-valued clausal formalism defined in [20–22] and to partial Max-SAT solvers [23].

References

1. Watson, J., Beck, J., Howe, A., Whitley, L.: Toward an understanding of local search cost in job-shop scheduling (2001)
2. Iwama, K., Kambayashi, Y., Miyano, E.: New bounds for oblivious mesh routing. In: European Symposium on Algorithms. (1998) 295–306
3. Zhang, Y., Zha, H., Chao-Hsien, C., Ji, X., Chen, X.: Towards inferring protein interactions: Challenges and solutions. In: EURASIP Journal on Applied Signal Processing. (2005)
4. Li, C.M., Manyà, F., Planes, J.: New inference rules for max-sat. *Journal of Artificial Intelligence Research* (2007) To appear.
5. Heras, F., Larrosa, J.: New inference rules for efficient max-sat solving. In: AAAI. (2006)
6. Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for max-sat solving. In: *ijcai07*. (2007)
7. Xing, Z., Zhang, W.: Maxsolver: an efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence* **164**(1-2) (2005) 47–80
8. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted csp. In: *IJCAI*. (2003) 239–244
9. Li, C.M., Manyà, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In: CP-2005, Springer LNCS 3709 (2005) 403–414
10. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In: AAAI. (2006) 86–91
11. Wallace, R., Freuder, E.: Comparative studies of constraint satisfaction and davis-putnam algorithms for maximum satisfiability problems. In: *Cliques, Colouring and Satisfiability*. (1996) 587–615
12. Wallace, R.J.: Directed arc consistency preprocessing. In: *Constraint Processing, Selected Papers, London, UK, Springer-Verlag* (1995) 121–137
13. Larrosa, J., Meseguer, P., Schiex, T.: Maintaining reversible dac for max-csp. *Artif. Intell.* **107**(1) (1999) 149–163
14. de Givry, S., J. Larrosa, P.M., Schiex, T.: Solving max-sat as weighted CSP. In: CP. (2003) 363–376
15. Shen, H., Zhang, H.: Study of lower bounds functions for max-2-sat. In: AAAI 2004. (2004) 185–190
16. Alsinet, T., Manyà, F., Planes, J.: Improved exact solver for weighted Max-SAT. In: SAT-2005, Springer LNCS 3569 (2005) 371–377
17. de Givry, S.: Singleton consistency and dominance for weighted csp. In: CP 2004, Workshop on Preferences and Soft Constraints. (2004)
18. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted csp. In: *IJCAI 2003*. (2003)
19. Xing, Z., Zhang, W.: Maxsolver: an efficient exact algorithm for (weighted) maximum satisfiability. *Artif. Intell.* **164**(1-2) (2005) 47–80
20. Béjar, R., Manyà, F.: Solving combinatorial problems with regular local search algorithms. In: LPAR’99, Springer LNAI 1705 (1999) 33–43
21. Béjar, R., Hähnle, R., Manyà, F.: A modular reduction of regular logic to classical logic. In: *ISMVL’01*. (2001) 221–226
22. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with boolean variables. In: SAT-2004, (Revised Selected Papers), Springer LNCS 3542 (2004) 1–15
23. Argelich, J., Manyà, F.: Exact Max-SAT solvers for over-constrained problems. *Journal of Heuristics* **12**(4–5) (2006) 375–392