Within-Problem Learning for Efficient Lower Bound Computation in Max-SAT Solving[*]

Han Lin
Department of Computer Science
Sun Yat-sen University
Guangzhou 510275, China
linhan095@gmail.com

Kaile Su[†]
Key Laboratory of High Confidence
Software Technologies (Peking University)
Ministry of Education, Beijing, China
Institute for Integrated and Intelligent Systems
Griffith University, Brisbane, Australia
sukl@pku.edu.cn

Chu-Min Li
MIS
Université de Picardie Jules Verne
33 Rue St. Leu
80039 Amiens, France
chu-min.li@u-picardie.fr

Abstract

This paper focuses on improving branch-and-bound Max-SAT solvers by speeding up the lower bound computation. We notice that the existing propagation-based computing methods and the resolution-based computing methods, which have been studied intensively, both suffer from several drawbacks. In order to overcome these drawbacks, we propose a new method with a nice property that guarantees the increment of lower bounds. The new method exploits within-problem learning techniques. More specifically, at each branch point in the search-tree, the current node is enabled to inherit inconsistencies from its parent and learn information about effectiveness of the lower bound computing procedure from previous nodes. Furthermore, after branching on a new variable, the inconsistencies may shrink by applying unit propagation to them, and such process increases the probability of getting better lower bounds. We graft the new techniques into *maxsatz* and the experimental results demonstrate that the new solver outperforms the best state-of-the-art solvers on a wide range of instances including random and structured ones.

Introduction

In recent years there has been an increasing interest in studying the Max-SAT problem. On one hand, more new applications of Max-SAT solvers have been found to areas as diverse as machine learning (Yang, Wu, & Jiang 2007), circuit debugging (Safarpour *et al.* 2007), and bioinformatics (Zhang *et al.* 2006). On the other hand, remarkable progress has been achieved in designing and implementing efficient exact Max-SAT solvers (Xing & Zhang 2005; Larrosa & Heras 2006; Larrosa, Heras, & de Givry 2008; Heras, Larrosa, & Oliveras 2008; Lin & Su 2007; Li, Manyà, & Planes 2006; 2007; Darras *et al.* 2007; Pipatsrisawat

& Darwiche 2007). The Max-SAT Evaluation 2006[1] and 2007[2] advanced and witnessed such progress.

The most common approach for solving Max-SAT is based on a branch and bound (BnB) algorithm, which is a general scheme for NP-hard optimization problems. Results from the Max-SAT Evaluation indicate that BnB solvers are generally faster than other approaches. The key factor that makes BnB solvers efficient is to compute lower bounds of good quality. In (Li, Manyà, & Planes 2005; 2006), unit propagation and failed literal detection, called propagation-based methods, are exploited to compute lower bounds. These procedures are able to identify more disjoint inconsistent subformulas for a given formula, thus improve previous lower bounds significantly. However, they still have large overheads. First, due to the lack of mechanism to maintain these inconsistent subformulas, such inconsistencies may be rediscovered again and again by the descendent nodes. Second, at a search node, if the lower bound obtained is less than the upper bound, the lower bound computation effort is completely wasted.

Another method, called resolution-based method, to compute lower bound is using inference rules to simplify formulas and make the conflict explicit (Larrosa & Heras 2006; Li, Manyà, & Planes 2007).This method may avoid the above problems by applying rules to the inconsistent subformulas. In this case, the explicit empty clauses make the lower bound in the subsequent search incremental, since the inconsistent subformulas involved are not encountered again and the empty clauses are inherited by the descendent nodes. Unfortunately, this method is limited because the inference rules cannot be applied unless a formula matches some special patterns. Moreover, applying rules may be time and space consuming because more new clauses have to be added to the current formula and removed upon backtracking. So, although complete resolution has been proposed by (Bonet, Levy, & Manyà 2007), only practically efficient rules are implemented in existing solvers.

In this paper, we attempt to propose a new algorithm for Max-SAT, which aims to address all the above problems. For this purpose, the new lower bound computation method in our algorithm must satisfy at least three requirements:

---

[1]http://www.iiia.csic.es/~maxsat06/
[2]http://www.maxsat07.udl.es/

First, it computes lower bounds in an incremental way; second, it can be applied to all types of inconsistent subformulas; third, it can be implemented efficiently.

The basic idea is to utilize the so-called within-problem learning techniques, which mean, at the current node, exploiting information gathered by other nodes of the search-tree to boost the whole depth-first search. This kind of techniques are widely used in SAT and CSP study. For example, *clause learning*, a well-known within-problem learning method, has been incorporated in the modern SAT (Zhang *et al.* 2001) and Max-SAT solvers (Argelich & Manyà 2007). In our algorithm, each node of the search tree learns inconsistencies from its parent, and eliminates the clauses which are reductant in the current inconsistencies. Moreover, the failed literal detection may be time consuming. Thus, our algorithm also enables each node of the search tree to learn the information about what the effectiveness of the failed literal detection is for the already explored nodes, and decide whether such procedure should be invoked. We graft these new techniques into the Max-SAT solver $maxsatz$ (Li, Manyà, & Planes 2006; 2007), and the experimental results demonstrate that our new techniques enhance the effectiveness.

The paper is organized as follows. In the next section, we introduce some necessary background and related work. Then we present our within-problem learning algorithms formally. We have implemented new Max-SAT solvers called $incmaxsatz$ by building on $maxsatz$. Experimental results demonstrating the performance of our solvers are presented next followed by some final conclusions.

Preliminaries and Related Work

In propositional logic, a variable x may take values $true$ or $false$. A *literal* is either a variable x or its negation $\bar{x}$. A *clause* $C = l_1 \vee l_2 \vee \ldots \vee l_k$ is a disjunction of literals. A *CNF formula* is a conjunction of clauses, which is usually represented as a set of clauses $\{C_1, C_2, \ldots, C_m\}$. For a formula F over a variable set V, an *assignment* is a mapping from V' to $\{0, 1\}$. The assignment is complete if $V' = V$; otherwise it is partial. An assignment satisfies a clause if it satisfies at least one literal in the clause, and satisfies a CNF formula if it satisfies all clauses. The *instantiation* of a formula F by forcing literal l to be $true$, denoted by $F[l]$, produces a new formula by removing all clauses containing l, and removing $\bar{l}$ from each clause where it occurs. This procedure is also called the application of *one-literal rule*. A formula is *inconsistent* if it cannot be satisfied by any assignment.

Given a CNF formula, the Max-SAT problem is to find an assignment that maximizes the number of satisfied clauses, or equivalently, minimizes the number of unsatisfied clauses.

The basic BnB algorithm for Max-SAT solving behaves as follows: Given a formula F, BnB traverses the search tree representing the space of all possible assignments in a depth-first manner. At each node, BnB computes the lower bound (LB), which is the underestimation of the minimum number of unsatisfied clauses if the current partial assignment is extended to a complete one. Then LB is compared with the upper bound (UB), which is the minimum number

of unsatisfied clauses found so far for a complete assignment. If $LB \geq UB$, BnB backtracks to a higher level in the search tree; otherwise, an unassigned variable is chosen to assign a value and the search continues. After the entire space is explored, the value UB takes is the minimum number of unsatisfied clauses in F.

As mentioned in the previous section, LB can be computed by exploiting unit propagation (UP) and failed literals detection (FL) to search for disjoint inconsistent subformulas. We briefly describe these two procedures as follows.

- UP (Li, Manyà, & Planes 2005): Given a formula F, UP maintains a set *UNIT*, which contains the unit clauses derived so far, and repeatedly applies one-literal rule, for each unit clause in *UNIT*, to simplify F until no more unit clause or an empty clause is derived. If an empty clause is derived, the set of clauses that used to derive this empty clause forms an inconsistent subformula IS. In order to identify other disjoint inconsistent subformulas, IS is removed from F, and the iterative process continues. If no more empty clause can be derived, then LB is equal to the number of empty clauses in F plus the number of inconsistent subformulas identified.

- FL (Li, Manyà, & Planes 2006): Given a formula F, FL applies unit propagation to $F \cup \{x\}$ and $F \cup \bar{x}$ for every variable x occurring in F. If these two processes both derive an empty clause, then $IS_x \cup IS_{\bar{x}} \setminus \{x, \bar{x}\}$ is an inconsistent subformula, where IS_x and $IS_{\bar{x}}$ are the inconsistent subformulas detected in $F \cup \{x\}$ and $F \cup \bar{x}$ respectively.

Note that in (Li, Manyà, & Planes 2006), FL is invoked after UP at every node unless UP leads to backtracking. In our algorithm, at each node, whether FL is executed depends on the effect FL had on the nodes which have been explored. Another new solver called *maxsatz14icss* (Darras *et al.* 2007), which memorizes inconsistencies containing at most 5 clauses and no branching variables, will be compared with our work in the following sections.

Within-problem Learning Algorithms

Learning from One's Own Parent

In classical SAT solving, the solver utilizes unit propagation to simplify a formula, so the propagation procedure at each node of the search tree is completely different from that at its parent node. While in Max-SAT solving, unit propagation is exploited to compute bounds. Since such computation does not change the formula, the propagation procedure at each node is very close to its parent. Thus, computing lower bounds at each node from scratch will waste a lot of time to re-apply unit propagation to the same clauses. This also happens in failed literal detection. Moreover, applying unit propagation from scratch may cause another problem: The lower bound may decrease as the search-depth increases, i.e., as more variables are instantiated.

Example 1 *Let* $F = \{x_1, \bar{x_1} \vee x_2, \bar{x_1} \vee \bar{x_2} \vee x_3, \bar{x_3} \vee x_4, \bar{x_2} \vee \bar{x_4}, x_5, \bar{x_5} \vee x_2, \bar{x_5} \vee x_4, \bar{x_5} \vee x_6, \bar{x_6} \vee x_7, \bar{x_6} \vee x_7, \bar{x_7} \vee x_8, \bar{x_8} \vee x_9, \bar{x_5} \vee \bar{x_8} \vee \bar{x_9}\}$. At first, $UNIT = \{x_1, x_5\}$.*

x_1 is propagated first, and we get an inconsistent subformula $IS_1 = \{x_1, \bar{x}_1 \vee x_2, \bar{x}_1 \vee \bar{x}_2 \vee x_3, \bar{x}_3 \vee x_4, \bar{x}_2 \vee \bar{x}_4\}$. Then x_5 is propagated, and another inconsistent subformula $IS_2 = \{x_5, \bar{x}_5 \vee x_6, \bar{x}_6 \vee x_7, \bar{x}_7 \vee x_8, \bar{x}_8 \vee x_9, \bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9\}$ is obtained. Suppose that we branch next on variable x_1. After assigning $true$ to x_1, we get $F' = \{x_2, \bar{x}_2 \vee x_3, \bar{x}_3 \vee x_4, \bar{x}_2 \vee \bar{x}_4, x_5, \bar{x}_5 \vee x_2, \bar{x}_5 \vee x_4, \bar{x}_5 \vee x_6, \bar{x}_6 \vee x_7, \bar{x}_7 \vee x_8, \bar{x}_8 \vee x_9, \bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9\}$ and new unit clause x_2 is added to $UNIT$ such that $UNIT = \{x_5, x_2\}$. In the following, we show that the previous lower bound computation methods will decrease LB.

- $maxsatz$[3]: According to the unit clause selection heuristic in $maxsatz$, x_5 will be propagated before x_2, because x_5 is "older" than x_2. After propagating x_5 on F', new unit clauses x_2, x_4 and x_6 are added to $UNIT$, and then after propagating x_2 and x_4, we derive an empty clause and obtain an inconsistent subformula $IS' = \{\bar{x}_2 \vee \bar{x}_4, x_5, \bar{x}_5 \vee x_2, \bar{x}_5 \vee x_4\}$. After excluding IS' from F', the remaining clause-set is $\{x_2, \bar{x}_2 \vee x_3, \bar{x}_3 \vee x_4, \bar{x}_5 \vee x_6, \bar{x}_6 \vee x_7, \bar{x}_7 \vee x_8, \bar{x}_8 \vee x_9, \bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9\}$, in which no addition inconsistency can be detected. Thus, the new lower bound $LB' = 1$, which is less than LB obtained at its parent node.

- $maxsatz14_{icss}$: According to the inconsistent subformula storage heuristic in $maxsatz14_{icss}$, IS_1 is not stored because the branching variable x_1 occurs in it, and neither is IS_2 because the number of clauses in it exceeds five. The following processes are the same as in the $maxsatz$ case.

As is known, lower bound computation plays an important role in reducing the size of the BnB search tree by making LB increase quickly so as to equal or exceed UB. So it would be better to avoid, when the search-depth increases, decreasing LB in lower bound computation. To the best of our knowledge, however, no previous method provides a mechanism to guarantee the increment of lower bounds, i.e., to guarantee that the lower bound at every node, except for the root, will be greater than or equal to the lower bound at its parent. In fact, there is a simple way to address this problem. At each node of the search-tree, we can memorize all the inconsistent subformulas after they are identified, such that each child-node is able to inherit all the inconsistencies of its parent first and then exploits unit propagation to find new inconsistencies. This process is safe because of the following observation.

Proposition 1 If F is an inconsistent formula, so are both $F[x]$ and $F[\bar{x}]$ for any variable x in F.

This proposition trivially holds and guarantees that all inconsistencies of a node are still present in other nodes of the subtree rooted by this node. It is clear that memorizing the inconsistencies guarantees the increment of lower bounds.

Example 2 In Example 1, after detecting IS_1 and IS_2, we memorize them. We still choose x_1, and assign $true$ to it. Now IS_2 is left intact because it does not contain x_1, and

IS_1 becomes $\{x_2, \bar{x}_2 \vee x_3, \bar{x}_3 \vee x_4, \bar{x}_2 \vee \bar{x}_4\}$ (by applying the one-literal rule). It is obvious that IS_1 and IS_2 are still inconsistent and LB keeps the value 2.

Although the above method avoids re-computation and guarantees the increment of lower bound, it is still far from satisfactory because the inconsistent subformulas may contain reductant clauses, i.e, clauses whose removal will not make the subformula satisfiable.

Example 3 Let F be the formula in Example 1, and IS_1 and IS_2 detected and memorized as in the previous examples. Now suppose that we branch next on x_7 rather than x_1, and assign it $false$. Now IS_1 is left intact and IS_2 becomes $\{x_5, \bar{x}_5 \vee x_6, \bar{x}_6, \bar{x}_8 \vee x_9, \bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9\}$, in which the clauses $\bar{x}_8 \vee x_9$ and $\bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9$ are reductant.

In the above example, if IS_2 can be reduced to $\{x_5, \bar{x}_5 \vee x_6, \bar{x}_6\}$, $\bar{x}_8 \vee x_9$ and $\bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9$ may be used, together with other clauses, to form new inconsistent subformulas such that LB increases.

In order to reduce an inconsistent subformula, one may utilize the algorithms for extracting minimal unsatisfiable cores like in (Mneimneh *et al.* 2005). However, these techniques, for a BnB Max-SAT solver, is costly because the unsatisfiable cores should be extracted many times even at one search node. Fortunately, we may appeal to unit propagation again. Unit propagation also works here because detecting an inconsistent formula by unit propagation means proving its unsatisfiability by unit resolution[4], and we have the following proposition.

Proposition 2 Let F be an inconsistent formula. If the unsatisfiability of F can be proved by unit resolution, so can $F[x]$ and $F[\bar{x}]$ for any variable x in F.

Proposition 2 guarantees that the child-node can apply unit propagation to detect, hopefully smaller, inconsistent subformulas from the ones inherited from its parent.

Example 4 In Example 3, applying unit propagation to the inconsistent subformula $\{x_5, \bar{x}_5 \vee x_6, \bar{x}_6, \bar{x}_8 \vee x_9, \bar{x}_5 \vee \bar{x}_8 \vee \bar{x}_9\}$ will identify a smaller one $\{x_5, \bar{x}_5 \vee x_6, \bar{x}_6\}$.

Apparently, the new lower bound computation method can avoid re-computation and guarantee the increment of lower bounds. Furthermore, thanks to the unit propagation to reduce inconsistencies, the memorized inconsistent subformulas may become smaller and smaller as more and more variables are instantiated. This means that we are more likely to get more inconsistent subformulas and improve lower bounds.

We found that, from the empirical study, it is not a good choice to memorize inconsistencies when LB is much less than UB. An intuitive explanation for this phenomena is that "a child should not learn from his parent when his parent does not act as a good example". So in our algorithm, the learning is triggered when $\frac{LB}{UB}$ reaches some ratio α where $0 \leq \alpha \leq 1$. The best value of α is instance-dependent, nevertheless we simply set $\alpha = 0.3$ and $\alpha = 0.8$ for Max-2-SAT and Max-3-SAT respectively. The empirical results show that such setting seems good enough.

---

[3]No efficient inference rules in (Li, Manyà, & Planes 2007) can be applied to IS_1 and IS_2 as they both contain a ternary clause.

[4]Unit resolution states that from l and $\bar{l} \vee D$, where l is a literal and D is a disjunction of literals, we can derive D.

Algorithm 1: IncMax

---

1 **IncMax(**F**,** UB**,** LB**,** ISS**,** α**)**

  Input: A CNF formula F, an upper bound UB, a lower
      bound LB, a set of disjoint inconsistent
      subformulas ISS and a constant ratio α.

  Output: The minimum number of unsatisfied clauses in
      F.

2 **begin**

3   **if** $F = \emptyset$ **or** F *only contains empty clauses* **then**

4   **return** $\#Empty(F)$

5    $F \leftarrow applyRules(F)$

6   **if** $LB < UB * \alpha$ **then**

7      $C \leftarrow identifyISS(F)$

8      $LB \leftarrow \#Empty(F) + \#C$

9   **else if** $LB < UB$ **then**

10   **foreach** *inconsistent subformula* $S \in ISS$ **do**

11   remove clauses of S from F

12      $C \leftarrow identifyISS(F)$

13      $LB \leftarrow \#Empty(F) + \#ISS + \#C$

14   **foreach** *inconsistent subformula* $S \in ISS$ **do**

15   reinsert clauses of S into F

16      $ISS \leftarrow ISS \cup C$

17   **if** $LB \geq UB$ **then**

18   **return** ∞

19    $x \leftarrow selectVariable(F)$

20   **if** $LB \geq UB * \alpha$ **then**

21   **foreach** *inconsistent subformulas* $S \in ISS$ **do**

22   **if** S *contains* x **then**

23   remove S from ISS

24          $S' \leftarrow UP(S)$

25          $ISS \leftarrow ISS \cup \{S'\}$

26    $UB \leftarrow min(UB, IncMax(F[\bar{x}], UB, LB, ISS, \alpha))$

27   **return** $min(UB, IncMax(F[x], UB, LB, ISS, \alpha))$

28 **end**

Our algorithm for Max-SAT with the new method proposed in this section to compute lower bounds is described in Algorithm 1, called $IncMax$. Given a formula F and a ratio α, $IncMax$ is called as $IncMax(F, \#clauses, 0, \emptyset, \alpha)$. The algorithm begins with handling some trivial cases (lines 3-4). Then it applies inference rules (like in (Li, Manyà, & Planes 2007)) to F[5] (line 5). If $\frac{LB}{UB}$ does not reach α, the procedure $identifyISS$ exploits unit propagation to identify disjoint inconsistent subformulas in F, and LB is computed (lines 6-8). Otherwise, LB is computed in an incremental way (lines 9-16): Remove the clauses already in ISS in order to avoid re-computation; exploits unit propagation to compute LB in remaining clauses; and then reinsert the clauses in ISS into F. If LB reaches UB the algorithm backtracks, otherwise a variable x is selected to instantiate

---

[5]In our solver, as well as $maxsatz$, some inference rules are applied after detecting inconsistent subformulas. However, for simplicity, the pseudo codes do not reflect this case.

(lines 17-19). If x is contained by an inconsistent subformula S, unit propagation is exploited to reduce it (lines 20-25). Finally, the instance is solved by two recursive calls for the two sub-instances $F[\bar{x}]$ and $F[x]$ (lines 26-27). Note that when $\alpha = 1$, $IncMax$ is the same as the algorithm proposed by (Li, Manyà, & Planes 2007).

The correctness of $IncMax$ is guaranteed by Proposition 1 and 2. As for complexity, $IncMax$ only uses extra linear space to store ISS compared with the algorithm proposed by (Li, Manyà, & Planes 2007).

Learning from Past

So far, our algorithm has not incorporated failed literal detection (FL). This is because if an inconsistent subformula IS is identified by FL, we cannot guarantee that the unsatisfiable core can be extracted by unit propagation (UP) after instantiating variables in it. Although, of course, it can be extracted by FL, it seems costly because FL is more time-consuming than UP. However, FL is very powerful for pruning branches on some instances, especially on Max-3-SAT and Max-CUT ones (Li, Manyà, & Planes 2006). So we extend our algorithm to a new version, called $IncMax^*$, which exploits FL after UP to detect additional inconsistent subformulas, but such inconsistencies will not be inherited by the subsequent nodes.

Moreover, we found that, from empirical investigation, FL can rarely lead to backtrackings, e.g., only 5% of executions of FL making LB exceed UB, on some instances. Thus, $IncMax^*$ enables the search-node to learn information about the effect FL has on the previous nodes. More specifically, at each node, FL is executed only if

$$\#FL \leq \#SAMPLE \quad \text{or} \quad \frac{\#FAIL * LB}{\#FL * UB} \geq \beta \quad (1)$$

where $\#FL$ denotes the current total times of executions of FL, and $\#FAIL$ denotes the current number of failures (backtrackings) led by FL. The condition (1) makes sure that for the first $\#SAMPLE$ times, FL is always executed. After then, whether FL is executed depends on what percentage of the former executions have led to backtrackings and how close is LB to UB. Note that, for an input formula, parameters β and $\#SAMPLE$ are automatic determined according to the number of variables n, the number of clauses m, and the maximum clause length k. Since FL seems powerful when $k \geq 3$, we set $\beta = 0.2$ and $\beta = 0.3$ when $k \geq 3$ and $k \leq 2$ respectively. For $\#SAMPLE$, ideally, it should be proportional to the size of the entire search tree. However, it is difficult to determine such size before search. So we set $\#SAMPLE$ to be $n * m/10$ and $n * m/100$ when $k \geq 3$ and $k \leq 2$ respectively.

Experimental Investigation

We have implemented $IncMax$ and $IncMax^*$ in two new Max-SAT solvers called *incmaxsatz* and *incmaxsatz+fl*, which were accomplished by extending the Max-SAT solvers *maxsatz* and *maxsatz+fl* respectively. We conducted an experimental investigation to compare our solvers with the following state-of-the-art solvers.
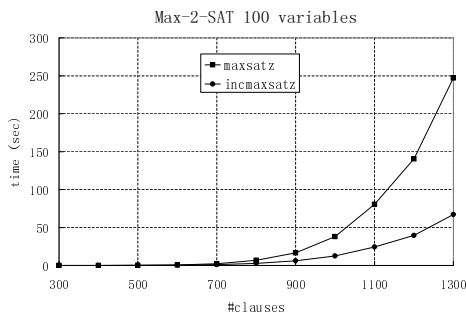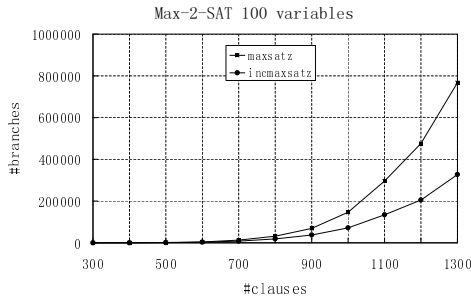
Figure 1: Mean time for Max-2-SAT

Figure 2: Mean number of branches for Max-2-SAT

Figure 3: Mean time for Max-3-SAT
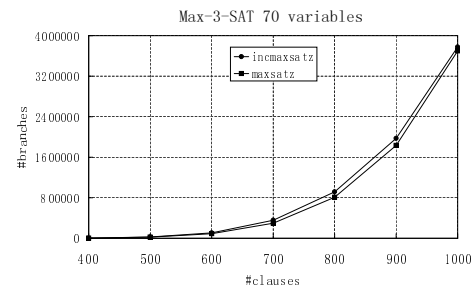
Figure 4: Mean number of branches for Max-3-SAT

- *maxsatz*[6] (Li, Manyà, & Planes 2007): a branch and bound solver which exploits unit propagation and some sophisticated inference rules to compute lower bounds.

- *maxsatz+fl*[7] (Li, Manyà, & Planes 2006; 2007): a solver extending *maxsatz* by incorporating failed literal detection. We used the evaluation version which turned out to be the best performing solver on unweighted instances in the Max-SAT Evaluation 2006 and 2007.

- *maxsatz14icss* (Darras *et al.* 2007): a solver, built on *maxsatz+fl*, using a heuristic to memorize inconsistencies.

- *Toolbar3.0*[8] (Larrosa & Heras 2006; Larrosa, Heras, & de Givry 2008): a Max-SAT solver which exploits resolution rules and weighted CSP solving techniques extensively.

- *MiniMaxSat*[9] (Heras, Larrosa, & Oliveras 2008): a new solver which incorporates many current SAT techniques, like clause learning and two-watched literal scheme.

All the benchmarks we used are from the Max-SAT Evaluation 2006 and 2007. Executions are run on a 1.7 Ghz Pentium 4 computer with 512 Mb of RAM under Linux.

In the first experiment we evaluated, on random 100-variable Max-2-SAT instances, the lower bound computation method by "learning from one's own parent". The num-

ber of clauses ranged from 300 to 1300. Figure 1 and Figure 2 show that the new method obtains better lower bounds than *maxsatz*. As the number of clauses increases, the power of the new method is clearer.

As for random 70-variable Max-3-SAT instances, the lower bounds obtained by our method seem close to *maxsatz* (see Figure 4). However, since the lower bound computing procedure is more efficient due to sparing re-computation time, *incmaxsatz* still outperforms *maxsatz* (see Figure 3).

On random 140-variable Max-2-SAT instances, we compared *incmaxsatz+fl* with all other solvers. The results (see Figure 5) demonstrate that *incmaxsatz+fl* is consistently better than others, especially when the number of clauses is large. On 80-variable Max-3-SAT instances, *incmaxsatz+fl* is as good as *maxsatz+fl*, and faster than others (Figure 6). As for structured instances (Table 1), *incmaxsatz+fl* is the best performing one on most classes of benchmarks.
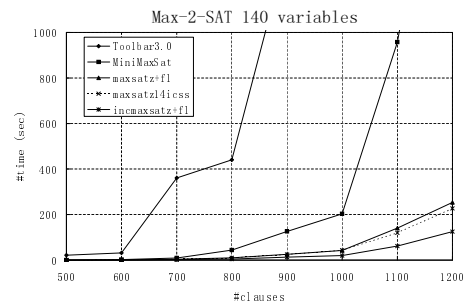
Figure 5: Mean time for Max-2-SAT

---

[6] The source code is downloadable from `http://web.udl. es/usuaris/m4372594/soft/maxsatz.tgz`.

[7] `http://web.udl.es/usuaris/m4372594/soft/ maxsatz-evaluation`

[8] `http://mulcyber.toulouse.inra.fr/`

[9] `http://www.lsi.upc.edu/~fheras/docs/m. tar.gz`

Benchmarks	#Instances	Toolbar3.0	MiniMaxSat	maxsatz+fl	maxsatz14icss	incmaxsatz+fl
brock	12	124.33(12)	66.51(12)	18.47(12)	17.87(12)	**15.37(12)**
c-fat	7	209.59(5)	0.23(5)	0.11(5)	0.11(5)	**0.09(5)**
hamming	6	1102.95(3)	834.74(3)	243.19(3)	**209.37(3)**	218.07(3)
johnson	4	238.46(3)	254.95(3)	62.29(3)	58.29(3)	**49.93(3)**
keller	2	27.67(2)	18.92(2)	8.59(2)	6.84(2)	**6.02(2)**
p_hat	12	177.06(12)	77.39(12)	21.79(12)	21.32(12)	**19.80(12)**
san	11	141.07(7)	980.18(10)	366.78(11)	371.16(11)	**332.53(11)**
sanr	4	591.05(4)	354.76(4)	97.10(4)	92.44(4)	**77.17(4)**
random-maxcut	40	98.89(40)	23.28(40)	8.13(40)	**7.60(40)**	7.90(40)
spinglass	5	10.67(2)	**2.19(3)**	63.45(3)	38.40(3)	43.49(3)

Table 1: Mean time on structured benchmarks. The number of solved instances (within one hour) is displayed in brackets, and the best solver is highlighted in bold text.
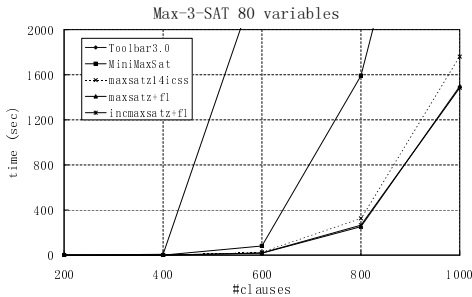

Figure 6: Mean time for Max-3-SAT

Conclusions and Future Work

We have presented within-problem learning techniques, based on the similarity (Propositions 1 and 2) among the nodes of the search tree, for improving lower bound computation in Max-SAT solving. We implemented Max-SAT solvers, which can be considered as generalized versions of *maxsatz*. The experimental investigation demonstrates that the new solvers are very competitive.

As future work we plan to extend the within-problem techniques to weighted Max-SAT, partial Max-SAT and even other optimization problems. The techniques are promising because we believe that the similarity, among nodes of a search tree, commonly exists in many problems.

References

Argelich, J., and Manyà, F. 2007. Partial max-sat solvers with clause learning. In *SAT-2007*, 28–40.

Bonet, M.; Levy, J.; and Manyà, F. 2007. Resolution for Max-SAT. *Artificial Intelligence* 171(8-9):606–618.

Darras, S.; Dequen, G.; Devendeville, L.; and Li, C. M. 2007. On inconsistent clause-subsets for max-sat solving. In *CP-2007*, 225–240.

Heras, F.; Larrosa, J.; and Oliveras, A. 2008. Mini-MaxSAT: An Efficient Weighted Max-SAT solver. *Journal of Artificial Intelligence Research* 31:1–32.

Larrosa, J., and Heras, F. 2006. New inference rules for efficient max-SAT solving. *Proc. of AAAI-06, Boston, MA*.

Larrosa, J.; Heras, F.; and de Givry, S. 2008. A Logical Approach to Efficient Max-SAT solving. *Artificial Intelligence* 172(2-3):204–233.

Li, C.; Manyà, F.; and Planes, J. 2005. Exploiting Unit Propagation to Compute Lower Bounds in Branch and Bound Max-SAT Solvers. In *CP-2005*, 403–414.

Li, C.; Manyà, F.; and Planes, J. 2006. Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In *AAAI-2006*, 86–91.

Li, C.; Manyà, F.; and Planes, J. 2007. New Inference Rules for Max-SAT. *Journal of Artificial Intelligence Research* 30:321–359.

Lin, H., and Su, K. 2007. Exploiting inference rules to compute lower bounds for max-sat solving. In *IJCAI-2007*, 2334–2339.

Mneimneh, M.; Lynce, I.; Andraus, Z.; Marques-Silva, J.; and Sakallah, K. 2005. A Branch-and-Bound Algorithm for Extracting Smallest Minimal Unsatisfiable Formulas. In *SAT-2005*. Springer.

Pipatsrisawat, K., and Darwiche, A. 2007. Clone: Solving weighted max-sat in a reduced search space. In *AI-2007*, 223–233.

Safarpour, S.; Mangassarian, H.; Veneris, A.; Liffiton, M.; and Sakallah, K. 2007. Improved Design Debugging using Maximum Satisfiability. In *FMCAD-07*.

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

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence* 171(2-3):107–143.

Zhang, L.; Madigan, C. F.; Moskewicz, M. W.; and Malik, S. 2001. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD-2001*, 279–285.

Zhang, Y.; Zha, H.; Chu, C.; and Ji, X. 2006. Towards Inferring Protein Interactions: Challenges and Solutions. *EURASIP Journal on Applied Signal Processing* 2006(1):56–56.