Detecting Disjoint Inconsistent Subformulas for Computing Lower Bounds for Max-SAT *

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

Felip Manyà
IIIA, CSIC
Campus UAB, 08193 Bellaterra, Spain
felip@iiia.csic.es

Jordi Planes
DIEI, Universitat de Lleida
Jaume II 69, E-25001 Lleida, Spain
jplanes@diei.udl.es

Abstract

Many lower bound computation methods for branch and bound Max-SAT solvers can be explained as procedures that search for disjoint inconsistent subformulas in the Max-SAT instance under consideration. The difference among them is the technique used to detect inconsistencies. In this paper, we define five new lower bound computation methods: two of them are based on detecting inconsistencies via a unit propagation procedure that propagates unit clauses using an original ordering; the other three add an additional level of forward look-ahead based on detecting failed literals. Finally, we provide empirical evidence that the new lower bounds are of better quality than the existing lower bounds, as well as that a solver with our new lower bounds greatly outperforms some of the best performing state-of-the-art Max-SAT solvers on Max-2SAT, Max-3SAT, and Max-Cut instances.

Introduction

The Max-SAT problem for a CNF formula ϕ is the problem of finding an assignment of values to variables that minimizes the number of unsatisfied clauses in ϕ.

In recent years we have seen considerable progress on the performance of Max-SAT solvers. Modern exact solvers compute optimal solutions much faster than solvers existing just five years ago; for example, the speedups are up to three orders of magnitude for random Max-2SAT instances with just 100 variables.

The most competitive exact Max-SAT solvers (Alsinet, Manyà, & Planes 2005; de Givry *et al.* 2003; Li, Manyà, & Planes 2005; Shen & Zhang 2004; Xing & Zhang 2005) implement variants of the following branch and bound (BnB) schema: Given a CNF formula ϕ, BnB explores the search tree that represents the space of all possible assignments for ϕ in a depth-first manner. At every node, BnB compares the upper bound (UB), which is the best solution found so far for a complete assignment, with the lower bound (LB), which is the sum of the number of clauses unsatisfied by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if the current partial assignment is completed. If $LB \geq UB$ the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If $LB < UB$, the algorithm tries to find a better solution by extending the current partial assignment by instantiating one more variable. The solution to Max-SAT is the value that UB takes after exploring the entire search tree.

In this paper we focus on the study of lower bounds computation methods for BnB Max-SAT solvers and, in particular, on defining underestimations of good quality that can be computed efficiently. We explain such methods as procedures that search for disjoint inconsistent subformulas in a Max-SAT instance. The difference among them is the technique used to detect inconsistencies. On the one hand, the bigger the number of detected inconsistencies, the better the quality of the lower bound. On the other hand, as a lower bound is computed at each node of the search tree, the detection of inconsistencies should be performed efficiently.

We start by giving some preliminary definitions and reviewing the most relevant state-of-the-art lower bound computation methods. We then define five new lower bound computation methods: two of them are based on detecting inconsistencies via a unit propagation procedure that propagates unit clauses using an original ordering; the other three add an additional level of forward look-ahead based on detecting failed literals. Finally, we provide empirical evidence that the new lower bounds are of good quality, as well as that a solver with our new lower bounds greatly outperforms some of the best performing state-of-the-art Max-SAT solvers on Max-2SAT, Max-3SAT, and Max-Cut instances.

Notation and Definitions

In propositional logic a variable x_i may take values 0 (for false) or 1 (for true). A literal l_i is a variable x_i or its negation $\neg x_i$. A clause is a disjunction of literals, and a CNF formula is a *multiset* of clauses. An assignment of truth values to the propositional variables satisfies a literal x_i if x_i takes the value 1 and satisfies a literal $\neg x_i$ if x_i takes the value 0, satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. The empty clause is denoted by $\Box$, and is unsatisfied by any assignment.

*Research partially supported by projects TIN2004-07933-C03-03 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*. The first author is partially supported by National 973 Program of China under Grant No. 2005CB321900. The second author is supported by a grant *Ramón y Cajal*.
Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Related Work

The lower bound computation method has a dramatic impact on the performance of any Max-SAT solver. The simplest method consists of just counting the number of clauses unsatisfied by the current partial assignment. This method was implemented by Borchers & Furman (1999).

One step forward is to incorporate an underestimation of the number of clauses that will become unsatisfied if the current partial assignment is extended to a complete assignment. The most basic method was defined by Wallace & Freuder (1996):

$$\texttt{LB}(\phi) = \#unsat + \sum_{x \text{ occurs in } \phi} min(ic(x), ic(\neg x)),$$

where ϕ is the CNF formula associated with the current partial assignment, $\#unsat$ is the number of empty clauses derived so far, and $ic(x)$ ($ic(\neg x)$) —inconsistency count of x ($\neg x$)— is the number of clauses that become unsatisfied if the current partial assignment is extended by fixing x to true (false); in other words, $ic(x)$ ($ic(\neg x)$) coincides with the number of unit clauses of ϕ that contain $\neg x$ (x). That method can be explained telling that the underestimation of the lower bound is the number of disjoint inconsistent subformulas formed by two complementary unit clauses.

Binary clauses can also contribute to the underestimation of the lower bound using the Directional Arc Consistency (DAC) count notion defined by Wallace (1995) for Max-CSP. The DAC count of a value of the variable x in ϕ is the number of variables which are inconsistent with that value of x. For example, if ϕ contains clauses $x \vee y$, $x \vee \neg y$, and $\neg x \vee y$, the value 0 of x is inconsistent with y, meaning that when 0 is assigned to x, the lower bound should be incremented by one. Note that value 0 of y is also inconsistent with x. These two inconsistencies are not disjoint and cannot be summed. Wallace defined a direction from x to y, so that only the inconsistency for value 0 of x is counted. After defining a direction between every pair of variables sharing a constraint, one computes the DAC for all values of x by checking all variables to which a direction from x is defined. The DAC notion of Wallace considers the next underestimation:

$$\sum_{x \text{ occurs in } \phi} (min(ic(x), ic(\neg x)) + min(dac(x), dac(\neg x)),$$

where $dac(x)$ ($dac(\neg x)$) is the DAC count of the value 1(0) of x. In (Wallace 1995), all directions are statically defined, so that $dac(x)$ and $dac(\neg x)$ can be computed in a preprocessing step for every x and do not need to be recomputed during search. Larrosa, Meseguer, & Schiex (1999) improved this by introducing reversible DAC, which searches for better directions to obtain a better LB at every step of search. A further improvement of DAC is the additional incorporation of inconsistencies contributed by disjoint subsets of variables, based on particular variable partitions (Larrosa & Meseguer 2002). Max-CSP techniques were applied to Max-SAT in (de Givry *et al.* 2003).

The most remarkable improvements to the previous lower bounds, dealing with more than two literals per clause, are the star rule (Shen & Zhang 2004; Alsinet, Manyà, & Planes 2004) and UP (Li, Manyà, & Planes 2005).

In the star rule, the underestimation of the lower bound is the number of disjoint inconsistent subformulas of the form $\{l_1, \ldots, l_k, \neg l_1 \vee \cdots \vee \neg l_k\}$. The star rule, when $k = 1$, is the method based on inconsistency counts.

In UP, the underestimation of the lower bound is the number of disjoint inconsistent subformulas that can be detected by applying unit resolution.[1] UP finds inconsistent subformulas as follows: It maintains a queue Q that contains the unit clauses that have been derived so far, and applies unit propagation[2] considering the unit clauses in the ordering of Q (i.e.; older unit clauses are preferred to more recent unit clauses). Once a contradiction is detected, UP analyzes the resolution steps performed and identifies, as an inconsistent subformula, a multiset of clauses that are able to derive the detected contradiction via unit resolution. Using technologies developed in modern SAT solvers such as Satz (Li & Anbulagan 1997), UP can be implemented efficiently.

UP subsumes the inconsistent count method based on unit clauses and the star rule. Its effectiveness for producing a good lower bound can be illustrated with the following example: Let ϕ be a CNF formula containing the clauses $x_1, \neg x_1 \vee x_2, \neg x_1 \vee x_3, \neg x_2 \vee \neg x_3 \vee x_4, x_5, \neg x_5 \vee x_6, \neg x_5 \vee x_7, \neg x_6 \vee \neg x_7 \vee \neg x_4$. UP easily detects that inconsistent subset, with 8 clauses and 7 variables, in time linear in the size of the formula. Note that this subset is not detected by any of the lower bounds described above, except for the variable partition based approach of Larrosa & Meseguer (2002) in the case that the 7 variables are in the same partition.

It is also worth to mention a lower bound for Max-2SAT, called LB4, that was defined by Shen & Zhang (2004), which is similar to UP: they detect disjoint inconsistent subformulas in Max-2SAT instances via linear unit resolution. Finally, we mention the lower bound computation defined by Xing & Zhang (2005), which detects disjoint inconsistent subformulas with a method based on linear programming.

We have reviewed the methods that improve the lower bound by computing underestimations. Another approach consist of applying inference rules that allow to transform a Max-SAT instance ϕ into an equivalent Max-SAT instance ϕ' that contains more empty clauses than ϕ. Inference rules preserving the equivalence among Max-SAT instances can be found, for instance, in (Alsinet, Manyà, & Planes 2005; Larrosa & Heras 2005).

Five New Lower Bounds

We first propose two new lower bounds, called UP* and UP^S, which improve UP by using better orderings for propagating unit clauses in unit propagation. Then, we propose three new lower bounds, called UP$_{FL}$, UP$^*_{FL}$ and UP$^S_{FL}$,

[1] Unit resolution states that from l and $\neg l \vee D$, where l is a literal and D is a disjunction of literals, we can derive the resolvent D.

[2] Unit propagation is the repeated application of the one-literal rule until reaching a contradiction or a saturation state. Given a CNF formula ϕ with a unit clause l, the one-literal rule deletes all the clauses containing l and removes all the occurrences of $\neg l$.

which are, respectively, extensions of UP, UP* and UP^S incorporating the detection of failed literals.

Lower Bounds Improving UP

UP gives an underestimation of the number of disjoint inconsistent subformulas in a CNF formula ϕ using unit propagation, which means that (i) each inconsistent subformula contains at least one unit clause and, therefore, the number of detected inconsistencies is bounded by the number of unit clauses in ϕ; and (ii) clauses in an inconsistent subformula cannot be used to derive other inconsistent subformulas.

In order to improve the underestimation of UP, we need to find disjoint inconsistent subformulas containing as few unit clauses as possible, leaving more unit clauses in the remaining formula to derive further inconsistent subformulas. For the same reason, each inconsistent subformula should also contain as few non-unit clauses as possible. That is the motivation of defining better orderings than the one implemented in UP for propagating unit clauses in unit propagation. As a result, we provide two new lower bounds: UP* and UP^S.

UP* maintains two queues: Q_1 and Q_2. When UP* starts to search for an inconsistent subformula, Q_1 contains all the unit clauses of the CNF formula under consideration (more recently derived unit clauses are at the end of Q_1), and Q_2 is empty. The unit clauses derived during the application of unit propagation are stored in Q_2, and unit propagation does not use any unit clause from Q_1 unless Q_2 is empty.

UP^S stores all unit clauses in a stack S instead of in a queue Q.

Example 1 *Let ϕ_1 be the Max-SAT instance $\{x_1, x_2, x_3, \neg x_1 \vee x_4, \neg x_1 \vee x_5, \neg x_4 \vee \neg x_5, \neg x_1 \vee \neg x_2 \vee \neg x_3, x_1 \vee \neg x_2\}$. We show that UP detects exactly one inconsistent subformula while UP* and UP^S are able to detect two inconsistent subformulas.*

- *UP: Initially, $Q = [x_1, x_2, x_3]$. When x_1 is propagated, unit clauses x_4 and x_5 are added to Q ($Q = [x_2, x_3, x_4, x_5]$), clause $x_1 \vee \neg x_2$ is removed, and clause $\neg x_1 \vee \neg x_2 \vee \neg x_3$ becomes $\neg x_2 \vee \neg x_3$. When x_2 is propagated, $\neg x_2 \vee \neg x_3$ becomes $\neg x_3$, which is added to Q ($Q = [x_3, x_4, x_5, \neg x_3]$). When x_3 is propagated, the empty clause is derived. The inconsistent subformula detected by UP is $\{x_1, x_2, x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3\}$. The remaining clauses $\{\neg x_1 \vee x_4, \neg x_1 \vee x_5, \neg x_4 \vee \neg x_5, x_1 \vee \neg x_2\}$ do not contain any unit clause and, therefore, UP stops.*
- *UP*: Initially, $Q_1 = [x_1, x_2, x_3]$. When x_1 is propagated, unit clauses x_4 and x_5 are added to Q_2 ($Q_2 = [x_4, x_5]$), clause $x_1 \vee \neg x_2$ is removed, and clause $\neg x_1 \vee \neg x_2 \vee \neg x_3$ becomes $\neg x_2 \vee \neg x_3$. We then propagate x_4 and derive $\neg x_5$, which is added to Q_2 ($Q_2 = [x_5, \neg x_5]$). When x_5 is propagated, the empty clause is derived. The first inconsistent subformula detected is $\{x_1, \neg x_1 \vee x_4, \neg x_1 \vee x_5, \neg x_4 \vee \neg x_5\}$. Observe that UP* consumed exactly one unit clause from the input formula. Next, UP* detects another contradiction in the remaining clauses: $\{x_2, x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3, x_1 \vee \neg x_2\}$. Now, $Q_1 = [x_2, x_3]$. When x_2 is propagated, unit clauses x_1 is added to Q_2 ($Q_2 = [x_1]$) and $\neg x_1 \vee \neg x_2 \vee \neg x_3$ becomes $\neg x_1 \vee \neg x_3$. When x_1 is propagated, unit clauses $\neg x_3$ is added to Q_2 ($Q_2 = [\neg x_3]$). When $\neg x_3$ is propagated, the empty clause is derived. The second inconsistent subformula is $\{x_2, x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3, x_1 \vee \neg x_2\}$.*
- *UP^S:Initially, $S = [x_3, x_2, x_1]$ (we assume x_1 is at the bottom of the stack). When x_3 is propagated, clause $\neg x_1 \vee \neg x_2 \vee \neg x_3$ becomes $\neg x_1 \vee \neg x_2$, and $S = [x_2, x_1]$. When x_2 is propagated, unit clauses $\neg x_1$ and x_1 are added to S ($S = [x_1, \neg x_1, x_1]$). When x_1 is propagated, the empty clause is derived. The first inconsistent subformula detected is $\{x_2, x_3, \neg x_1 \vee \neg x_2 \vee \neg x_3, x_1 \vee \neg x_2\}$. Next, UP^S derives another contradiction from the remaining clauses: $\{x_1, \neg x_1 \vee x_4, \neg x_1 \vee x_5, \neg x_4 \vee \neg x_5\}$. Now, $S = [x_1]$. When x_1 is propagated, unit clauses x_4 and x_5 are added to S ($S = [x_5, x_4]$). When x_5 is propagated, unit clauses $\neg x_4$ is added to S ($S = [\neg x_4, x_4]$). When $\neg x_4$ is propagated, the empty clause is derived. The second inconsistent subformula is $\{x_1, \neg x_1 \vee x_4, \neg x_1 \vee x_5, \neg x_4 \vee \neg x_5\}$.*

Example 1 suggests that one of the drawbacks of UP is that it consumes unit clauses from the input formula that could be avoided, which is a direct consequence of the ordering in which unit clauses are propagated.

Example 2 *Let ϕ_2 be the Max-SAT instance $\{x_1, \neg x_1 \vee x_2, \neg x_1 \vee x_3, \neg x_2 \vee \neg x_3, \neg x_1 \vee x_4, \neg x_4 \vee x_5, \neg x_5 \vee x_6, \neg x_6 \vee x_7, \neg x_7 \vee \neg x_8, \neg x_7 \vee \neg x_9, x_8 \vee x_9\}$. We show that, in this case, UP^S consumes more clauses (not necessarily unit clauses) than UP* when detecting inconsistent subformulas.*

- *UP^S: Initially, $S = [x_1]$. When x_1 is propagated, unit clauses x_2, x_3, and x_4 are added to S ($S = [x_4, x_3, x_2]$). When x_4 is propagated, unit clause x_5 is added to S ($S = [x_5, x_3, x_2]$). When x_5 is propagated, unit clause x_6 is added to S ($S = [x_6, x_3, x_2]$). When x_6 is propagated, unit clause x_7 is added to S ($S = [x_7, x_3, x_2]$). When x_7 is propagated, unit clauses $\neg x_8$ and $\neg x_9$ are added to S ($S = [\neg x_9, \neg x_8, x_3, x_2]$). When $\neg x_9$ is propagated, unit clause x_8 is added to S. ($S = [x_8, \neg x_8, x_3, x_2]$). When x_8 is propagated, the empty clause is derived. The inconsistent subformula detected by UP^S is $\{x_1, \neg x_1 \vee x_4, \neg x_4 \vee x_5, \neg x_5 \vee x_6, \neg x_6 \vee x_7, \neg x_7 \vee \neg x_8, \neg x_7 \vee \neg x_9, x_8 \vee x_9\}$, which contains 8 clauses.*
- *UP*: Initially, $Q_1 = [x_1]$ and Q_2 is empty. When x_1 is propagated, unit clauses x_2, x_3, and x_4 are added to Q_2 ($Q_2 = [x_2, x_3, x_4]$). When x_2 is propagated, unit clause $\neg x_3$ is added to Q_2 ($Q_2 = [x_3, x_4, \neg x_3]$). When x_3 is propagated, the empty clause is derived. The inconsistent subformula detected by UP* is $\{x_1, \neg x_1 \vee x_2, \neg x_1 \vee x_3, \neg x_2 \vee \neg x_3\}$, which contains 4 clauses.*

Example 2 suggests that UP^S tends to find larger inconsistent subformula than UP*; i.e., UP^S can consume more clauses than UP* to derive an empty clause. This is so because UP^S, when there are several possibilities of deriving an empty clause from a unit clause, just finds the first derivation, while it can be shown that UP* always finds the shortest derivation. UP* makes one step in each possible derivation in parallel, stopping all derivations when the first empty clause is found. In other words, UP^S performs a depth-first search while UP* performs a breadth-first search.

Example 3 *Let clauses in ϕ_2 be ordered as follows: $\{x_1, \neg x_1 \vee x_4, \neg x_4 \vee x_5, \neg x_5 \vee x_6, \neg x_6 \vee x_7, \neg x_7 \vee \neg x_8, \neg x_7 \vee \neg x_9, x_8 \vee x_9, \neg x_1 \vee x_2, \neg x_1 \vee x_3, \neg x_2 \vee \neg x_3\}$.*

In this case, UP^S finds the shortest derivation to an empty clause, because the shortest derivation happens to be the first one. However, UP* always finds this derivation in the following way: initially, $Q_1 = [x_1]$ and Q_2 is empty. When x_1 is propagated, unit clauses x_4, x_2, and x_3 are added to Q_2 ($Q_2 = [x_4, x_2, x_3]$). When x_4 is propagated, unit clause x_5 is added to Q_2 ($Q_2 = [x_2, x_3, x_5]$). When x_2 is propagated, unit clause $\neg x_3$ is added to Q_2 ($Q_2 = [x_3, x_5, \neg x_3]$). When x_3 is propagated, the empty clause is derived.

Extending UP, UP* and UP^S with Failed Literal Detection

UP$_{FL}$, UP$^*_{FL}$ and UP$^S_{FL}$ are extension, respectively, of UP, UP* and UP^S, incorporating an additional level of forward look-ahead based on the detection of failed literals.[3]

Let ϕ be a Max-SAT instance, and let ϕ' be the formula resulting from ϕ after replacing every inconsistent subformula detected by UP* with an empty clause. Obviously, unit propagation in ϕ' cannot derive any additional empty clause. However, if unit propagation is applied to $\phi' \cup \{x\}$ and $\phi' \cup \{\neg x\}$, for any variable x occurring in ϕ', and produces an empty clause in each CNF formula (i.e.; x and $\neg x$ are failed literals in ϕ'), then $(\varphi_1 \cup \varphi_2) \setminus \{x, \neg x\}$ is an inconsistent subformula of ϕ', where φ_1 is the inconsistent subformula detected by UP* in $\phi' \cup \{x\}$, and φ_2 is the inconsistent subformula detected by UP* in $\phi' \cup \{\neg x\}$. That is a direct consequence of the following observation: We can produce a proof of $\neg x$ by applying resolution (not necessarily unit resolution) to $\varphi_1 \setminus \{x\}$, and a proof of x by applying resolution to $\varphi_2 \setminus \{\neg x\}$. If we put the two proofs together and resolve x and $\neg x$, we get a refutation from $(\varphi_1 \cup \varphi_2) \setminus \{x, \neg x\}$. Note that now (i) we only consider clauses of ϕ', and (ii) the refutation is not a *unit* resolution refutation; it is a resolution refutation.

Example 4 *Let ϕ' be $\{\Box, \Box, y \vee \neg x, \neg y \vee z, \neg y \vee \neg z, y \vee x\}$. If unit propagation is applied to $\phi' \cup \{x\}$, UP* detects the inconsistent subformula $\varphi_1 = \{x, y \vee \neg x, \neg y \vee z, \neg y \vee \neg z\}$, and if it is applied to $\phi' \cup \{\neg x\}$, UP* detects the inconsistent subformula $\varphi_2 = \{\neg x, \neg y \vee z, \neg y \vee \neg z, y \vee x\}$. Observe that a resolution refutation can be derived from $(\varphi_1 \cup \varphi_2) \setminus \{x, \neg x\} = \{y \vee \neg x, \neg y \vee z, \neg y \vee \neg z, y \vee x\}$.*

That result is exploited in the new lower bounds but, as introducing an additional level of look-ahead is time consuming, only a subset of the variables occurring in the CNF formula are used to detect failed literals. Note that, since a literal corresponds to a value of a variable, the detection of failed literals subsumes the DAC count for binary clauses.

We next describe in detail UP$^*_{FL}$, and assume that inconsistent subformulas in $\phi' \cup \{x\}$ and $\phi' \cup \{\neg x\}$ are detected via UP*. UP* was selected because it is better than UP and UP^S, but the previous result holds for UP and UP^S as well.

[3]The detection of failed literals is related to shaving in scheduling, and singleton local consistency in constraint satisfaction.

Let $Var(\phi')$ be the set of propositional variables occurring in ϕ' such that (i) they do not occur in unit clauses; and (ii) they have at least two positive occurrences and two negatives occurrences in binary clauses. UP$^*_{FL}$ detects, for each variable x in $Var(\phi')$, if x and $\neg x$ are both failed literals in ϕ'. Once an inconsistent subformula γ is detected, γ is replaced with an empty clause in ϕ', and the set of variables in which failed literals are searched for is updated taking into account the new CNF formula derived.

Variables occurring in unit clauses are not considered because they do not lead to a contradiction if UP* is applied to ϕ'. The fact of selecting variables with at least two positive occurrences and two negatives occurrences in binary clauses was determined empirically. These variables give at least two new unit clauses when they are set to a truth value.

UP$^*_{FL}$ computes, in general, tighter bounds (the total number of empty clauses in the resulting CNF formula) than UP* and, in the worst-case, it provides the same lower bound as UP*. It is also important to highlight some side effects of its application: (i) as soon as the new lower bound reaches the upper bound for some variable x, we can prune the current search subspace, and (ii) if the difference between the current lower bound and the upper bound is one and unit propagation in $\phi' \cup \{x\}$ ($\phi' \cup \{\neg x\}$) leads to an empty clause, then x can be set to false (true).

UP$_{FL}$ and UP$^S_{FL}$ are like UP$^*_{FL}$, except that they generate ϕ' from ϕ, respectively, with UP and UP^S.

Experimental Investigation

In the experimental investigation, we evaluated the impact of the different lower bounds defined in this paper, and compared a Max-SAT solver that implements UP$^*_{FL}$, which is our best performing lower bound, with the following solvers:

- SZ (Shen & Zhang 2005): a BnB Max-2SAT solver implementing the lower bound LB4.
- Toolbar2.2 (de Givry *et al.* 2003): a Max-SAT solver that encodes the input instance as a constraint network and solves that network with a state-of-the-art Max-CSP solver. We used version 2.2 with default parameters.
- MaxSolver (Xing & Zhang 2004): a BnB Max-SAT solver that applies a number of efficient inference rules. We used the 2nd release.
- Lazy (Alsinet, Manyà, & Planes 2005): a BnB Max-SAT solver with lazy data structures and a static variable selection heuristic. We used version 2.0.
- UP1.5 (Li, Manyà, & Planes 2005): a BnB Max-SAT solver implementing UP. We used version 1.5.

In the rest of this section, when we say solver UP, UP*, UP^S, UP$_{FL}$, UP$^*_{FL}$, and UP$^S_{FL}$ we refer to an improved version of the Max-SAT solver UP1.5 augmented with the lower bound computation method of the same name.

We provided the same initial upper bound to all the solvers, which was computed with a local search solver. All the experiments were performed on a Linux Cluster with 2GHz AMD Opteron processors with 1Gb of RAM.

As benchmarks we used randomly generated Max-2SAT instances and Max-3SAT instances, as well as Max-Cut

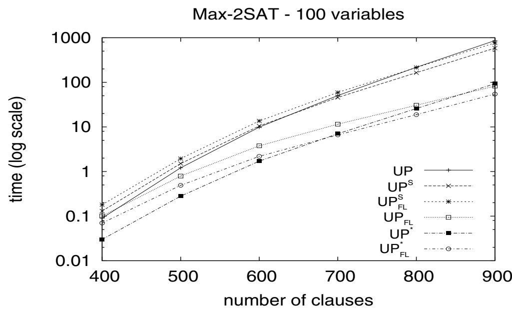

Figure 1: Mean time, in seconds, for Max-2SAT

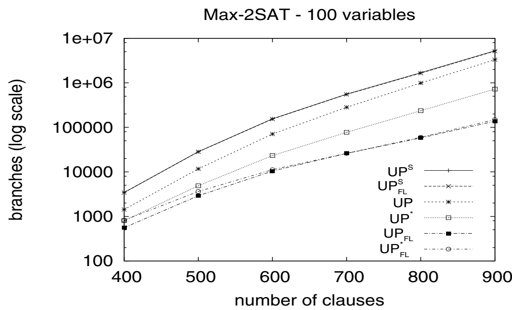

Figure 2: Mean number of branches for Max-2SAT

instances. Max-2SAT instances and Max-3SAT instances were created with the generator mwff.c developed by Bart Selman, which allows for duplicated clauses. For Max-Cut, we used the same encoding as Shen & Zhang (2004).

In the first experiment we evaluated, on random Max-2SAT instances, the lower bounds based on unit propagation. We solved sets of 100 instances with 100 variables; the number of clauses ranged from 400 to 900. The results obtained are shown in Figure 1. Along the horizontal axis is the number of clauses, and along the vertical axis is the mean time, in seconds, needed to solve an instance of a set. Figure 2 shows the mean number of branches of the proof tree. Notice that we use a log scale to represent both run-time and branches. Then, we compared UP$^*_{FL}$ with the Max-SAT solvers Lazy, MaxSolver, SZ, Toolbar2.2, UP and

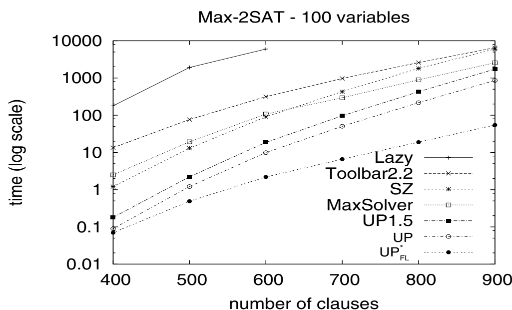

Figure 3: Comparison of Max-SAT solvers on Max-2SAT

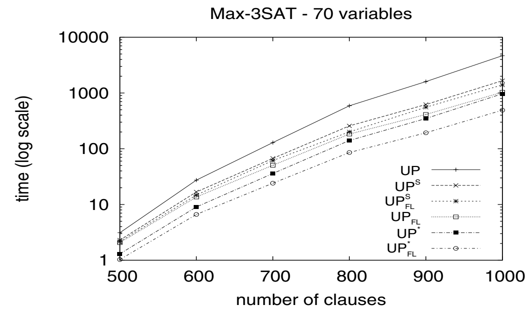

Figure 4: Mean time, in seconds, for Max-3SAT

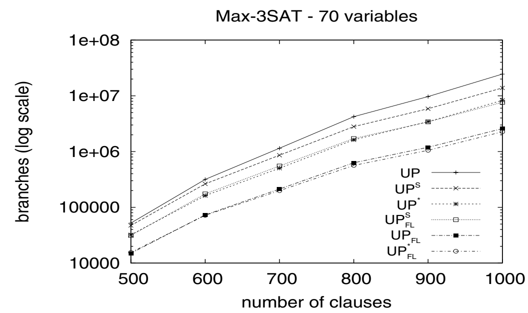

Figure 5: Mean number of branches for Max-3SAT

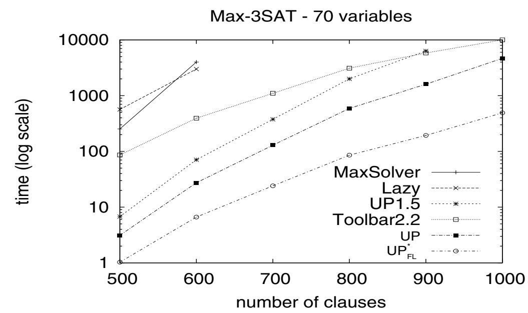

Figure 6: Comparison of Max-SAT solvers on Max-3SAT

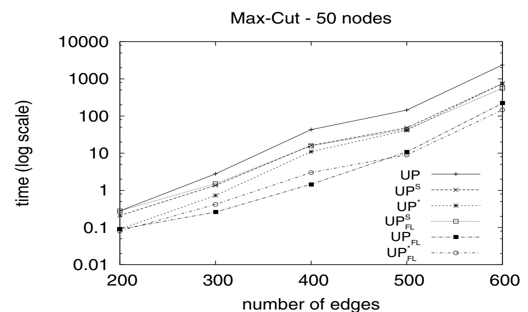

Figure 7: Mean time, in seconds, for Max-Cut

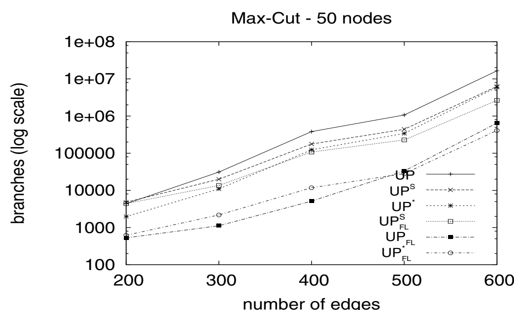

Figure 8: Mean number of branches for Max-Cut

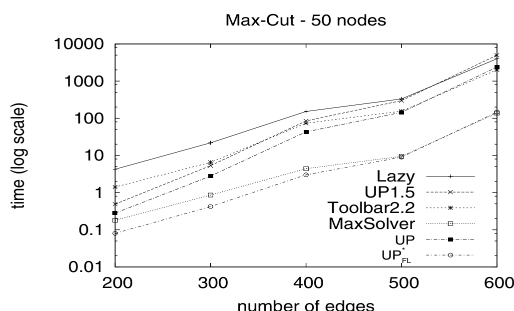

Figure 9: Comparison of Max-SAT solvers on MAX-CUT

UP1.5. Figure 3 shows the results obtained.

In the second experiment we evaluated the new lower bounds on random Max-3SAT instances. We solved sets of 100 instances with 70 variables; the number of clauses ranged from 500 to 1000. Figure 4 shows the mean time, in seconds, needed to solve an instance of a set, and Figure 5 shows the mean number of branches of the proof tree. We also compared UP^*_{FL} with the Max-SAT solvers Lazy, MaxSolver, Toolbar2.2, UP and UP1.5. Figure 6 shows the results obtained.

In the third experiment, whose results are shown in Figure 7 and Figure 8, we evaluated the new lower bounds on Max-Cut instances. We solved sets of 100 graphs with 50 vertices; the number of edges ranged from 200 to 600. Finally, we compared UP^*_{FL} with the Max-SAT solvers Lazy, MaxSolver, SZ, Toolbar2.2, UP and UP1.5. Figure 9 shows that comparison.

The experimental results provide evidence that UP^*_{FL} is the best lower bound introduced in this paper; for example, UP^*_{FL} is one order of magnitude faster than UP on the hardest Max-2SAT instances. The results also provide evidence that a solver implementing UP^*_{FL} slightly improves SZ and is superior to the rest of solvers on Max-Cut, and produces important speedups on Max-2SAT and Max-3SAT. In Figures 3 and 6, UP^*_{FL} is about 30 times faster than the second best solver on the hardest instances.

Conclusions

We contributed to the definition of good quality lower bounds in Max-SAT. We improved UP by using more efficient orderings in the propagation, and showed that UP-based lower bounds can be augmented with failed literal detection, obtaining tighter lower bounds.

The experimental results indicate that unit propagation, along with failed literal detection, gives an efficient method for computing good quality lower bounds if unit clauses are propagated following a suitable ordering. In particular, the ordering in UP^*_{FL} is the best option in our study, because it produces many more inconsistent subformulas than other orderings by consuming fewer unit and non-unit clauses when detecting an inconsistent subformula.

References

Alsinet, T.; Manyà, F.; and Planes, J. 2004. A Max-SAT solver with lazy data structures. In *IBERAMIA 2004*, 334–342.

Alsinet, T.; Manyà, F.; and Planes, J. 2005. Improved exact solver for weighted Max-SAT. In *SAT-2005*, 371–377.

Borchers, B., and Furman, J. 1999. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization* 2:299–306.

Larrosa, J., and Heras, F. 2005. Resolution in Max-SAT and its Relation to Local Consistency in Weighted CSPs. In *IJCAI-2005*, 193–198.

de Givry, S.; Larrosa, J.; Meseguer, P.; and Schiex, T. 2003. Solving Max-SAT as weighted CSP. In *CP-2003*, 363–376.

Larrosa, J., and Meseguer, P. 2002. Partition-based lower bound for Max-CSP. *Constraints* 7(3–4):407–419.

Larrosa, J.; Meseguer, P.; and Schiex, T. 1999. Maintaining reversible DAC for Max-CSP. *Artificial Intelligence* 107(1):149–163.

Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97*, 366–371.

Li, C. M.; 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.

Shen, H., and Zhang, H. 2004. Study of lower bound functions for max-2-sat. In *AAAI-2004*, 185–190.

Shen, H., and Zhang, H. 2005. Improving exact algorithms for max-2-sat. *Annals of Mathematics and Artificial Intelligence* 44:419–436.

Wallace, R., and Freuder, E. 1996. Comparative studies of constraint satisfaction and Davis-Putnam algorithms for maximum satisfiability problems. *Cliques, Coloring and Satisfiability*, volume 26. 587–615.

Wallace, R. J. 1995. Directed arc consistency preprocessing. In Springer LNCS 923, 121–137.

Xing, Z., and Zhang, W. 2004. Efficient strategies for (weighted) maximum satisfiability. In *CP-2004*, 690–705.

Xing, Z., and Zhang, W. 2005. An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence* 164(2):47–80.