Improved Exact Solver for the Weighted Max-SAT Problem

Adrian Kügel
Faculty of Engineering and Computer Sciences
Ulm University
Adrian.Kuegel@uni-ulm.de

Abstract

Many exact Max-SAT solvers use a branch and bound algorithm, where the lower bound is calculated with a combination of Max-SAT resolution and detection of disjoint inconsistent subformulas. We propose a propagation algorithm which improves the detection of disjoint inconsistent subformulas compared to algorithms previously used in Max-SAT solvers. We implemented this algorithm in our new solver akmaxsat and compared our solver with three solvers using unit propagation and restricted failed literal detection; these solvers are currently state-of-the-art on random Max-SAT instances. We also developed a lazy deletion data structure for our solver which speeds up lower bound calculation on instances with a high clauses-to-variables ratio. Our experiments show that our solver runs faster than the previously best solvers on randomly generated instances with a high clauses-to-variables ratio.

1 Introduction

The Max-SAT problem can be stated as follows: given a list of clauses $C_1, \ldots, C_m$, find an assignment of Boolean values to the variables $x_1, \ldots, x_n$ which satisfies the maximum number of clauses. The weighted Max-SAT problem is a generalization where each clause has a weight, and the sum of the weights of the satisfied clauses has to be maximized. Equivalently one can minimize the sum of the weights of the unsatisfied clauses, which is done in most Max-SAT solvers.

It is well known that the Max-SAT problem is NP-hard, e. g. Max-Clique and Max-Cut instances can be expressed as Max-SAT formulas. But Max-SAT has found applications in fields such as bioinformatics [11], electronic markets [10], sports scheduling [9] and routing [12]. Therefore a lot of research has gone into developing Max-SAT solvers.

Many competitive exact Max-SAT solvers use a branch and bound algorithm which operates on a binary tree, where each inner node corresponds to a partial assignment, and leaf nodes correspond to complete assignments. At every node of the search tree, a lower bound is calculated on the minimum number of unsatisfied clauses for any complete assignment which extends the current partial assignment. If the lower bound is at least as big as the best solution found so far, the subtree can be pruned, otherwise the partial assignment is extended by instantiating another variable. The quality of the lower bound function is very important, as it determines how much of the search tree can be pruned. One also has to consider, however, that there is a tradeoff between the quality of the lower bound and the efficiency of the lower bound function.

The simplest lower bound function just calculates the number of clauses unsatisfied by the current partial assignment. This can be improved by calculating an underestimation of the number of clauses which will become unsatisfied by extending the partial assignment to a complete assignment. Much progress has been made to compute good quality underestimations efficiently [6, 4, 7, 3, 8]. In this paper we present a generalized version of the algorithm in [7] which improves the quality of the underestimations. This algorithm is implemented in our solver *akmaxsat* which uses a lazy deletion data structure which speeds up lower bound calculation on instances with a high clauses-to-variables ratio.

This paper is organized as follows: in Sect. 2 we start with definitions, then in Sect. 3 we present our propagation algorithm, which improves detection of disjoint inconsistent subformulas. In Sect. 4

we describe the transformation rules which are used in our solver, and in Sect. 5 we describe the data structure of our solver. Finally, in Sect. 6 we show the results of experiments with our Max-SAT solver and three state-of-the-art solvers on benchmark instances of the Max-SAT evaluation 2009 and some randomly generated weighted Max-2-SAT instances.

2 Definitions

A *CNF formula* $\mathscr{F}$ is a conjunction of clauses consisting of literals from the set $\{x_1,\ldots,x_n\}\cup\{\bar{x}_1,\ldots,\bar{x}_n\}$, where $x_1,\ldots,x_n$ are variables which can take either the value true or false. A literal $\bar{x}_i$ is true if the variable x_i is false, and it is false otherwise. A *clause* C consists of a disjunction of literals and is written as $(l_1\vee l_2\vee\cdots\vee l_k)$. We call a clause *satisfied* if at least one of its literals is true, and we call it *unsatisfied* otherwise. An empty clause which contains no literal is denoted by $\square$ and is defined to be unsatisfied.

A *hard clause* is a clause which needs to be satisfied, whereas a *soft clause* specifies a clause which may be unsatisfied by the optimal assignment. In the SAT problem, all clauses are considered to be hard, whereas in the (weighted) Max-SAT problem we deal with only soft clauses. Instances of the (weighted) partial Max-SAT problem contain both soft and hard clauses, and we look for an assignment which satisfies all hard clauses and satisfies the maximum number of the soft clauses (or the soft clauses of maximum total weight in case of weighted formulas).

We define the size of a clause to be the number of literals it consists of. A clause of size 1 is called a *unit clause*. A CNF formula which consists only of clauses of size k is also called a k-SAT formula, the corresponding Max-SAT problem Max-k-SAT.

An *assignment* assigns each variable in a formula a truth value, and a *partial assignment* assigns truth values to a subset of the variables. A variable is *instantiated* by assigning it a truth value. Instantiating a variable yields a simplified formula without literals which have been assigned false, and without clauses which have become true. We denote by $\mathscr{F}[x]$ the resulting simplified formula after setting variable x to true, and by $\mathscr{F}[\bar{x}]$ the simplified formula after setting x to false.

3 Generalized Unit Propagation

Our algorithm is a generalization of unit propagation and the restricted version of failed literal detection which are currently used in Max-SAT solvers. Therefore we start by describing how unit propagation and failed literal detection work.

Unit propagation is based on the observation, that a unit clause can only be satisfied if the literal it consists of is set to true. In each step a unit literal l is selected and set to true, and consequently $\bar{l}$ is set to false. The literal $\bar{l}$ can therefore be removed from the clauses where it appears, possibly yielding new unit literals which can be used to continue this process. When an empty clause is derived, an inconsistent subset can be reconstructed by identifying the clauses which were used to derive the empty clause. Algorithm 1 shows a way how to do this.

At the beginning of the algorithm, the inconsistent subformula $\mathscr{F}'$ consists of the clause which has become empty during the unit propagation. In each step of the while loop it identifies if the propagated literal l was needed to derive the empty clause. If $\bar{l}$ does not occur in any clause in $\mathscr{F}'$, then the propagation of l did not help to derive the empty clause and therefore we do not need to add the clause C to $\mathscr{F}'$.

Failed literal detection temporarily adds a unit literal l to the formula and then uses unit propagation. If the empty clause can be derived, l is called a failed literal, and a subformula $\mathscr{F}_l$ can be extracted which provides a resolution proof of $\bar{l}$. If both l and $\bar{l}$ are failed literals, the subformula $\mathscr{F}_l\cup\mathscr{F}_{\bar{l}}$ forms an inconsistent subformula [7].

Algorithm 1 ReconstructInconsistentSubformula

Require: stack S of clauses used for unit propagation
Ensure: inconsistent formula $\mathscr{F}'$

```
F' := pop(S) /* the top of the stack is the clause which has become empty */
while S is not empty do
  C := pop(S)
  let l be the literal which has been propagated based on C
  if l̄ occurs in a clause in F' then
    F' := F' ∪ C
  end if
end while
return F'
```

We can generalize this algorithm. The idea is, that after having detected a failed literal l, we can add $\bar{l}$ to the formula and run unit propagation again. If this unit propagation does not yield an empty clause, we try to find another failed literal (but now in the current, simplified formula). This process can be extended until an empty clause is derived or no more failed literal can be found. Note that the algorithm will terminate, because in each propagation step the formula is simplified such that it does not contain the propagated literal or its negation anymore. An important part of the algorithm is that we need to keep track which clauses have been used to derive a unit literal.

In SAT-solvers like picosat similar algorithms are used, but they are even more general, as they do not only learn unit clauses, but also clauses of size > 1. In fact, generalized unit propagation could be seen as running a SAT-solver with unsatisfiable core detection, unit propagation and failed literal detection, but with the restrictions to do no branches and to learn only unit clauses. The failed literal detection proposed in [7] would mean a further restriction to start with failed literal detection until a failed literal is found and then do unit propagation starting with the complement of the failed literal until a conflict is found.

Algorithm 2 shows in pseudocode how the generalized unit propagation algorithm works. Whenever we run out of unit literals, we try to find a failed literal l in the current formula $\mathscr{G}$. For each variable, we check if the literal with the larger weight is a failed literal, where the weight is calculated as the number of clauses of size > 1 in which the literal occurs. We also tried to check the literal with smaller weight, but this seems to produce bigger inconsistent subformulas, which in turn leads to smaller lower bounds. The possible reason is that if we check the literal with larger weight and it is a failed literal, propagating $\bar{l}$ reduces the size of more clauses which in turn helps to derive an empty clause more quickly.

If we succeed in finding a failed literal l, we push the subformula $\mathscr{F}'_l$ of $\mathscr{F}$ on the stack which corresponds to the subformula in $\mathscr{G}$ providing a resolution proof of $\bar{l}$, propagate $\bar{l}$ and continue with generalized unit propagation. In case that no failed literal can be found, we stop. Note that unit propagation and failed literal detection as in [7] are special cases of our propagation algorithm.

When the generalized unit propagation succeeds in finding an empty clause, we can reconstruct the inconsistent subformula with the same algorithm used for unit propagation, the only difference is that the stack which is given as parameter to Algorithm 1 may contain sets of clauses instead of just single clauses. After extracting an inconsistent subformula, we can increase the lower bound by one (or by the minimum weight of clauses in $\mathscr{F}'$ in case of weighted formulas). We continue the search for other inconsistent subformulas in the remaining formula $\mathscr{F} \setminus \mathscr{F}'$. For weighted formulas, we determine the minimum clause weight of a clause in $\mathscr{F}'$ and subtract this weight from the weight of each clause in $\mathscr{F}'$ before continuing with generalized unit propagation.

Algorithm 2 GeneralizedUnitPropagation

Require: formula $\mathscr{F}$ on variables V
Ensure: inconsistent subformula $\mathscr{F}'$

$S := \emptyset$ /* is a stack of clause sets which have been used to propagate a literal */
$\mathscr{G} := \mathscr{F}$
stop := false
while stop = false **do**
  while $\mathscr{G}$ contains a unit literal **do**
    let C be a unit clause in $\mathscr{G}$ containing literal l
    let C' be the clause in $\mathscr{F}$ corresponding to C
    push(S, C'), $\mathscr{G} := \mathscr{G}[l]$
    if $\mathscr{G}$ contains an empty clause **then**
      let C_{empty} be the clause in $\mathscr{F}$ corresponding to the empty clause in $\mathscr{G}$
      push(S, C_{empty})
      $\mathscr{F}'$:= ReconstructInconsistentSubformula(S)
      return $\mathscr{F}'$
    end if
  end while
  stop := true
  for $x \in V$ **do**
    if weight(x) > weight($\bar{x}$) **then**
      $l := x$
    else
      $l := \bar{x}$
    end if
    if UnitPropagate($\mathscr{G} \cup l$) yields inconsistent subformula $\mathscr{F}_l$ **then**
      let $\mathscr{F}_l'$ be the subformula in $\mathscr{F}$ corresponding to $\mathscr{F}_l \setminus l$ in $\mathscr{G}$
      push(S, $\mathscr{F}_l'$), $\mathscr{G} := \mathscr{G}[\bar{l}]$
      stop := false
      break
    end if
  end for
end while

We will demonstrate our algorithm on the following CNF formula:

$$(x_1 \vee x_2 \vee x_6) \wedge (x_2 \vee \bar{x}_6) \wedge (\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2 \vee x_5) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_5)$$

Since there is no unit literal, unit propagation will stop immediately. The failed literal detection can only find the failed literal x_1, and $\bar{x}_1$ can be resolved from the subformula $(\bar{x}_1 \vee x_3) \wedge (\bar{x}_1 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4)$. In the next step, $\bar{x}_1$ is propagated, yielding the formula

$$(x_2 \vee x_6) \wedge (x_2 \vee \bar{x}_6) \wedge (\bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_5) \wedge (\bar{x}_2 \vee \bar{x}_5)$$

Another failed literal detection may find the failed literal x_2 e. g., and $\bar{x}_2$ is propagated. The formula now becomes $x_6 \wedge \bar{x}_6 \wedge (\bar{x}_3 \vee \bar{x}_4)$. The following unit propagation quickly derives the empty clause from $x_6 \wedge \bar{x}_6$.

In our solver we implemented generalized unit propagation with a few additional optimizations: we assign each variable a priority based on the product of the weights of the two corresponding literals. We

keep a list of variables which is initially sorted in non-increasing order by priority. We search failed literals in the order of the list. After each successful run of generalized unit propagation, we move the variables which correspond to failed literals detected during the generalized unit propagation to the end of the list (keeping their relative order). This makes sure that the next time generalized unit propagation is run, we will prefer variables which were not used in the previous step.

Another optimization works as follows: after running a failed literal detection which yields a failed literal l, we want to detect a literal as close to the conflict clause as possible whose propagation alone leads to the conflict (named first unique implication point in [13]). Obviously, such a unique implication point is a failed literal, too, and has a resolution proof which is a subset of the one we have already extracted. To find a unique (not necessarily the first) implication point, we use this efficient method: we determine the first literal l' (first in propagation order) which is propagated during failed literal detection for which $\bar{l}'$ occurs in more than one clause in the inconsistent subformula $\mathscr{F}_l$. This means after propagating l', it is the first time we could possibly have more than one literal on the propagation stack, and all clauses used for propagation so far must have been binary. We select l' as failed literal instead of l which allows us to shorten the inconsistent subformula $\mathscr{F}_l$ accordingly by removing all binary clauses which led to the propagation of l'.

Moreover we use the optimization of [6] and process unit literals in the order that they were encountered. We use generalized unit propagation at each node of the search tree, but we stop looking for further inconsistent subformulas as soon as the lower bound exceeds the optimum value found so far.

4 Max-SAT Resolution

Since lower bound computation takes much time, it is beneficial to transform the formula into a solution-equivalent simpler formula. In [4] it is shown how the SAT resolution rule can be extended to Max-SAT. The difference between SAT resolution and Max-SAT resolution is that we cannot just add the resolvent of two clauses to the formula, we need to replace the original clauses with the resolvent and some compensation clauses. Several Max-SAT resolution steps can be aggregated to yield transformation rules for specific clause sets.

The WMaxSatz solver [5] uses transformation rules which can be implemented efficiently as a byproduct of unit propagation or failed literal detection. This means that the transformation rules can be applied at each node of the search tree. We use the same transformation rules in our solver. In particular, these transformation rules are:

1. $l_1 \vee l_2, l_1 \vee \bar{l}_2 \Longrightarrow l_1$
2. $l_1, \bar{l}_1 \vee l_2, \bar{l}_2 \vee l_3, \ldots, \bar{l}_k \vee l_{k+1}, \bar{l}_{k+1} \Longrightarrow \square, l_1 \vee \bar{l}_2, l_2 \vee \bar{l}_3, \ldots, l_k \vee \bar{l}_{k+1}$ $(k \geq 0)$
3. $\bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3 \Longrightarrow \bar{l}_1, l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3$
4. $l_1, \bar{l}_1 \vee l_2, \bar{l}_2 \vee l_3, \ldots, \bar{l}_k \vee l_{k+1}, \bar{l}_k \vee l_{k+2}, \bar{l}_{k+1} \vee \bar{l}_{k+2} \Longrightarrow \square, l_1 \vee \bar{l}_2, l_2 \vee \bar{l}_3, \ldots, l_{k-1} \vee \bar{l}_k, l_k \vee \bar{l}_{k+1} \vee \bar{l}_{k+2}, \bar{l}_k \vee l_{k+1} \vee l_{k+2}$ $(k \geq 1)$

Since weighted clauses can be seen as aggregated unweighted clauses containing the same literals, we calculate the minimum weight $w_{\min}$ of a clause on the left hand side of a transformation rule, subtract $w_{\min}$ from the weight of all clauses on the left hand side, and each clause on the right hand side gets the weight $w_{\min}$.

5 The Lazy Deletion Data Structure

In the following paragraphs we describe the new lazy data structure that we use in our Max-SAT solver. Each Max-SAT instance consists of a list of clauses (and for weighted Max-SAT also of clause weights). First we analyze what kind of basic operations on the clauses we need:

1. Clause traversal: traverse all clauses in which literal l occurs.
2. Unit clause traversal: traverse all unit clauses.
3. Clause deletion: delete a clause (we want to keep only active clauses, which are not yet satisfied by the current partial assignment).
4. Clause insertion: insert a clause (either a new one derived by Max-SAT resolution, or reinsert old clauses when backtracking).
5. Assignment: assign a truth value to a variable.
6. Backtrack: undo the assignment of a truth value to the last variable instantiated.

We want to do all these operations efficiently from a theoretical and practical point of view, i. e. in amortized constant time per clause. In order to reach this objective, we keep additional data on top of the clauses: for each of the possible $2 \cdot n$ literals we keep an array of pointers to clauses of at least size two in which the literal occurs. In addition to that we keep an end pointer to the last clause pointer in each array. Also we keep for each literal the sum of the weights of the unit clauses consisting of that literal, and on top of that a linked list of literals with positive unit clause weight.

To support deletion and insertion, for each array we keep a timestamp which indicates when the array was traversed most recently. We store for each clause the literals it consists of, the current size of the clause, the clause weight, a reference counter, a special flag, a deletion flag, and a deletion timestamp. The reference counter indicates how many pointers to this clause exist in the pointer arrays. The special flag is true if the clause should be deleted when there are no more pointers referencing it, which can be checked with the reference counter. The delete flag is true if and only if one of the following conditions is met: the clause should be deleted, the weight of the clause is zero, the number of literals in the clause is less than two, or the clause is satisfied by the current partial assignment.

We distinguish between temporary deletions and permanent deletions. In any case, when a delete flag is set to true, we want to delete all clause pointers in the clause pointer arrays referencing the clause. To do a permanent deletion, we also want to delete the clause and its corresponding data. This is needed for clauses added by a transformation rule when we backtrack to a state before the application of the transformation rule, otherwise the memory consumption would grow proportionally to the search tree size. But we do not want to delete one of the original clauses permanently, since we have to restore them when we backtrack in the search tree. To be able to restore clauses, we keep a stack of clause pointers to clauses which have been deleted temporarily. We can then restore them in the reverse order of their deletion time. We will now describe how the six operations are implemented in our data structure.

The clause traversal works as follows: we traverse the array of literal l which contains the clause pointers to clauses where literal l occurs. When a clause pointer is encountered which points to a clause with a delete flag set to true, we delete the clause pointer. If in addition to that the special flag is true, we decrement the reference counter of the clause, and when it reaches 0 in this step, we delete the clause. The deletion of the clause pointer can be done in constant time, since we just replace the clause pointer with the last clause pointer in the array and decrement the end pointer by one.

The unit clause traversal is easy: we traverse the linked list of literals with positive unit clause weight.

For the clause deletion operation, we set the delete flag of the clause to true and store the current timestamp as deletion timestamp. In case we want to do a permanent deletion, we also set the special flag to true. Otherwise, we push the clause pointer on the stack of clauses which have to be restored later on. The deletion of clause pointers from the arrays is then done lazily during the traversal of the clause pointer arrays.

To insert a clause, we add a clause pointer to the clause pointer array of each literal which occurs in the clause. If we insert a new clause, we also add the clause itself to the list of clauses (together with the supporting data). Otherwise, we are inserting a clause which has been deleted before, so we need to set the delete flag to false. Note that we have to avoid to insert a clause pointer in an array if it is still there. This can happen if we restore a clause which has been temporarily deleted, but the lazy deletion has not yet happened. If the traversal timestamp of an array is smaller than the deletion timestamp of the clause, the clause pointer must still be in the array, otherwise if the traversal timestamp is bigger, the clause pointer must have been deleted, and we just add the clause pointer at the end of the array.

For the assignment operation, without loss of generality assume that literal l is set to true, and $\bar{l}$ is set to false. We use the first operation to traverse all clauses which contain l and set the delete flag of these clauses to true. We keep the clause pointers to the newly fulfilled clauses in the clause pointer array of l, because these clauses have to be restored when we backtrack later. Then we traverse all clauses which contain $\bar{l}$, and for each clause we decrement the size value of the clause. If the clause becomes a unit clause after the assignment, we set the delete flag to true and add the weight of the clause to the unit weight of the corresponding literal. If the unit weight was zero before, we also add the literal to the linked list of literals with a positive unit weight. Again we can keep the clause pointers in the clause pointer array of $\bar{l}$ and use them later to restore clauses which have become unit clauses in this step and therefore may have been deleted from another clause pointer array.

The backtrack operation works in the opposite way: we traverse the clauses which had been fulfilled (and thus possibly temporarily deleted) by the assignment and use the insert operation to reinsert clause pointers into clause pointer arrays where needed. Then we traverse the clauses containing the literal which had been set to false and increase the size value of the clauses. If a clause was a unit clause consisting of a literal l, we subtract the weight of the clause from the unit weight of l, and we use the insert operation to reinsert a clause pointer into the clause pointer array of l if needed. If the unit weight becomes zero, we remove l from the linked list of literals with a positive unit weight.

To show that the insert operation is correct, we prove the following invariant:

Invariant 5.1. *After each of the six operations, the clause pointer array of an unassigned literal l consists of pointers to all clauses in the list of clauses which contain literal l and either have a delete flag set to false, or the delete flag was set to true some time after the most recent traversal of the array.*

Proof. We can assume that the invariant holds before each operation. We prove the invariant by analyzing the cases when it can become false.

1. A clause is added: we use the insert operation, which makes sure that the invariant stays true.

2. A delete flag changes from true to false: again, we apply the insert operation. Since we know that the invariant holds before the execution of the insert operation, the decision if a clause pointer should be added to an array is correct, so the array of each literal occurring in the clause will contain a clause pointer to the clause after the application of the insert operation.

3. The pointer array of a literal is traversed: we remove all clause pointers pointing to clauses with a delete flag set to true, therefore the invariant will still hold.

□

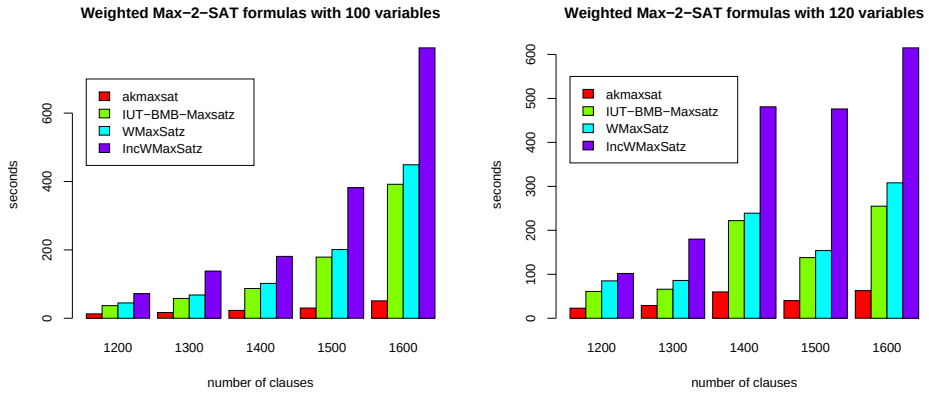

Figure 1: Average runtime on 10 weighted Max-2-SAT formulas of each type

This lazy deletion scheme is new to the best of our knowledge. The WMaxSatz solver, for example, does not remove clause pointers from the arrays, it just skips clauses with a delete flag during traversals of a clause pointer array. The advantage of our data structure is that when the formula becomes smaller by assigning variables, the lower bound calculation takes less time. We have some small overhead, however, therefore if each list of clause pointers is already small in the beginning, we do not save time. But the performance gain from using our data structure grows with the clauses-to-variables ratio of a formula.

6 Experimental Results

For the experiments in this section we used the version of our solver akmaxsat which we submitted to the 2010 Max-SAT evaluation. It is not optimized for unweighted formulas and allows clause weights up to 2^{63}. We have tested our solver together with the WMaxSatz solver [5], the IUT-BMB-Maxsatz solver (or IUT-BCMB-WMaxsatz solver for weighted instances; for simplicity we will call these two solvers IUT-Maxsatz solver from now on) and the IncMaxSatz (IncWMaxSatz) solver [8] on several sets of benchmark instances from the 2009 Max-SAT evaluation [2]. The other solvers were among the best solvers in the random categories of the 2009 Max-SAT evaluation.

In addition to the benchmark instances we randomly generated weighted Max-2-SAT instances with 100 variables or 120 variables, respectively. The number of clauses was chosen as 1200, 1300, 1400, 1500 or 1600, and we generated 10 random formulas of each type. Each weight was chosen randomly between 1 and 10 (like the comparable random instances of the 2009 Max-SAT evaluation). These instances were used to show how the different solvers scale on weighted Max-SAT formulas for increasing clauses-to-variables ratio. We also submitted these instances to the Max-SAT evaluation 2010 where they were used in the weighted random category.

Like in the Max-SAT evaluation, we used a timeout of 30 minutes for each instance; if a solver didn't terminate within 30 minutes, we regarded the instance as unsolved. We evaluated the number of solved instances and the average runtime on the solved instances for each solver. We ran the experiments on a node of the bwGRiD [1] which provides two Intel Harpertown quad-core CPUs with 2.83 Ghz and 8GB RAM each. The installed operating system was Scientific Linux.

Figure 1 shows the average runtime of the Max-SAT solvers on our randomly generated weighted Max-2-SAT instances. We can see that our solver akmaxsat outperforms the other solvers on these instances, and that it scales better for increasing clauses-to-variables ratio. Our solver is even 7.6 times faster on average than the second fastest solver IUT-Maxsatz on the formulas with 100 variables and 1600 clauses.

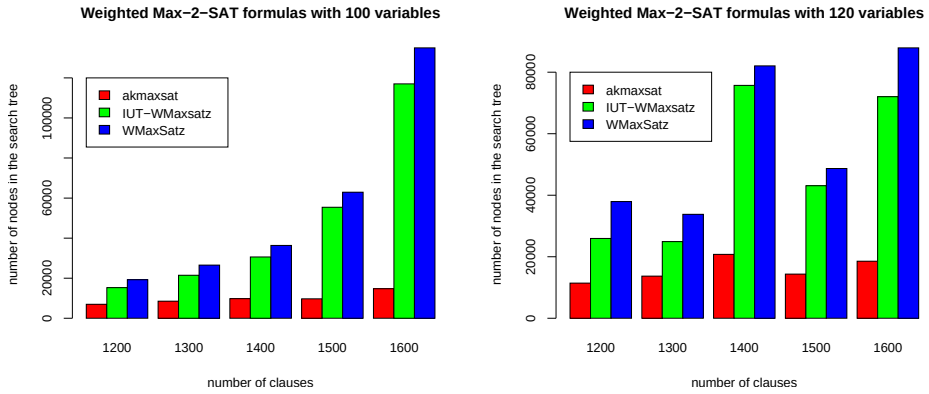

Figure 2: Average search tree size on 10 weighted Max-2-SAT formulas of each type

Figure 3: Unweighted random category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncMaxSatz
highgirth/4SAT	1089.85 (17)	1003.89 (15)	963.26 (9)	**504.05 (25)**
highgirth/5SAT	0.23 (25)	0.23 (25)	0.36 (25)	**0.13 (25)**
max2sat/100v	10.41 (50)	**8.61 (50)**	24.52 (50)	47.75 (50)
max2sat/120v	75.96 (50)	**64.65 (50)**	183.54 (50)	274.78 (47)
max2sat/140v	252.33 (49)	**236.19 (50)**	424.54 (48)	690.35 (38)
max3sat/60v	**60.12 (50)**	77.47 (50)	126.92 (50)	64.04 (50)
max3sat/70v	**226.34 (50)**	286.02 (50)	412.78 (49)	242.29 (50)
max3sat/80v	185.50 (50)	213.60 (50)	282.30 (49)	**178.59 (50)**

In order to explain the advantage of our solver, we calculated the average number of nodes in the search tree that each solver processes. Figure 2 shows the average search tree size of the different solvers on our randomly generated weighted Max-2-SAT instances. Note that the IncWMaxSatz solver does not print this information when processing an instance, therefore we could not include it in the figure.

We can see that our solver has a significantly smaller search tree size on weighted Max-2-SAT instances compared to the other solvers. Since we use the same branching and transformation rules in our solver as the WMaxSatz solver, the smaller search tree size must be caused by the generalized unit propagation algorithm, which seems to produce lower bounds of better quality which allow to prune more nodes of the search tree. We believe that our data structure plays an important role in the speed gain of our solver, too: although our lower bound calculation is computationally more expensive, the ratio of the search tree size of our solver to that of the other solvers is about the same as the ratio of the runtimes. It seems that for formulas with a sufficiently high clauses-to-variables ratio our data structure leads to speed gains which balance the additional time spent for lower bound calculation.

Figures 3 to 9 show the results of the solvers on several benchmark instance sets of the 2009 Max-

Figure 4: Unweighted crafted category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncMaxSatz
MAXCUT/DIMACS_MOD	**32.25 (52)**	52.09 (52)	95.47 (52)	37.75 (52)
MAXCUT/SPINGLASS	7.20 (3)	6.55 (3)	**3.08 (3)**	14.70 (3)
bipartite/maxcut-140-630-0.7	130.87 (50)	**117.23 (50)**	266.19 (49)	235.78 (50)
bipartite/maxcut-140-630-0.8	100.62 (50)	**86.61 (50)**	233.91 (50)	170.21 (50)

Figure 5: Weighted random category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncWMaxsatz
wmax2sat/hi	**1.42 (40)**	2.41 (40)	3.08 (40)	2.66 (40)
wmax2sat/lo	0.14 (40)	0.13 (40)	0.17 (40)	**0.07 (40)**
wmax3sat/hi	**60.06 (40)**	155.13 (40)	161.11 (40)	100.62 (40)
wmax3sat/lo	1.25 (40)	2.02 (40)	2.17 (40)	**1.16 (40)**

Figure 6: Weighted crafted category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncWMaxsatz
KeXu/frb	175.11 (14)	12.60 (9)	12.49 (9)	**63.16 (14)**
RAMSEY	7.74 (36)	21.64 (36)	10.09 (36)	**4.08 (36)**
WMAXCUT/DIMACS_MOD	**41.16 (60)**	77.28 (57)	77.48 (57)	114.19 (59)
WMAXCUT/SPINGLASS	**8.90 (4)**	26.84 (4)	27.69 (4)	17.80 (4)

Figure 7: Partial random category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncWMaxsatz
pmax2sat/hi	**5.07 (30)**	9.08 (30)	10.50 (30)	64.10 (30)
pmax2sat/me	**2.77 (30)**	8.15 (30)	4.52 (30)	15.01 (30)
pmax2sat/lo	0.63 (30)	**0.45 (30)**	0.50 (30)	0.99 (30)
pmax3sat/hi	66.10 (30)	**58.30 (30)**	60.57 (30)	107.03 (30)
pmax3sat/lo	0.68 (30)	0.34 (30)	0.38 (30)	**0.33 (30)**

Figure 8: Partial crafted category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncWMaxsatz
MAXCLIQUE/RANDOM	4.36 (96)	72.68 (84)	69.89 (84)	**1.78 (96)**
MAXCLIQUE/STRUCTURED	113.59 (36)	138.13 (19)	113.63 (25)	**101.80 (37)**
MAXONE/3SAT	0.98 (80)	123.32 (79)	129.69 (79)	0.25 (80)
MAXONE/STRUCTURED	207.73 (37)	70.84 (58)	71.24 (58)	120.99 (56)
PSEUDO/miplib	392.25 (3)	0.06 (2)	0.05 (2)	0.03 (2)
frb	483.62 (5)	0.00 (0)	0.00 (0)	**176.60 (5)**
min-enc/kbtree	150.71 (19)	202.15 (20)	**199.16 (20)**	495.46 (10)

Figure 9: Weighted partial random category

Instance set	akmaxsat	IUT-Maxsatz	WMaxSatz	IncWMaxsatz
wpmax2sat/hi	**14.96 (30)**	29.77 (30)	35.17 (30)	156.47 (30)
wpmax2sat/me	**7.09 (30)**	37.32 (30)	10.78 (30)	31.85 (30)
wpmax2sat/lo	1.46 (30)	**1.06 (30)**	1.18 (30)	1.39 (30)
wpmax3sat/hi	64.91 (30)	69.73 (30)	71.75 (30)	**55.06 (30)**
wpmax3sat/lo	0.55 (30)	0.36 (30)	0.44 (30)	**0.23 (30)**

SAT evaluation. For each instance set and each solver, we show in parentheses the number of solved instances and the average runtime on the solved instances in seconds. The data of the best performing solver for each instance set is printed in bold. We compare the performance of the solvers first by number of instances solved, in case of ties we compare the average runtimes on the solved instances. Since our solver does expensive lower bound calculations at each node of the search tree, it is not suitable for industrial instances and crafted instances with big formulas. Therefore we did not include the industrial categories and the weighted partial crafted category. Also we did not list instance sets where none of the tested solvers was able to solve any instance (for example the instance set JobShop of the partial crafted category).

Our solver performs quite well for the categories that we tested. Of the 37 total instance sets, it is the best performing solver among the tested solvers for 12 instance sets, and the second best performing solver for 16 instance sets.

In the partial crafted category there exists two instance sets (pseudo/miplib and MAXONE/STRUCTURED) where pm2 and SAT4J-Maxsat performed best in the Max-SAT evaluation 2009. Although pm2 was also the best solver in the evaluation for instance sets MAXONE/3SAT and frb, the IncWMaxsatz solver is much faster in our tests for these instances, so although we don't have exact comparison of runtimes we can be quite sure that IncWMaxsatz is really faster. For the instance sets MAXONE/STRUCTURED and pseudo/miplib we didn't highlight any of the tested solvers by printing its data in bold. The complete results can be found at www.uni-ulm.de/in/theo/mitarbeiter/kuegel.

The results of the Max-SAT evaluation 2010 support our own test results and show that akmaxsat can outperform state-of-the-art solvers. We submitted two versions of our solver to the evaluation: akmaxsat (the version which was used for the tests shown in this paper) and a version named akmaxsat_ls which does local search in the beginning to find an initial upper bound on the solution. akmaxsat_ls was the best solver in the unweighted random, the weighted (partial) random and the weighted (partial) crafted categories, and akmaxsat was the second-best solver for the weighted (partial) random and the weighted (partial) crafted categories.

The results of the evaluation also show that our solver is not competitive for industrial instances. Since industrial instances usually consist of big formulas with a low clauses-to-variables ratio, our lower bound calculation is too expensive, and the data structure is not helpful. An interesting question is if it is possible to use the branch and bound method also successfully on industrial instances. To the best of our knowledge, there is currently no branch and bound solver which is competitive with the best solvers on industrial instances; the most promising approach seems to be to use incremental lower bound calculations like in IncMaxSatz.

7 Conclusions

We have developed a propagation algorithm which can be used to improve the lower bound calculation of branch and bound Max-SAT solvers. We implemented it in our own Max-SAT solver and showed experimentally that it leads to a significant reduction of the number of nodes of the search tree that need to be traversed. We also presented evidence that our solver scales better than the other tested solvers for increasing clauses-to-variables ratio. We are confident that our propagation algorithm and the lazy deletion data structure can also improve other exact Max-SAT solvers.

8 Acknowledgments

We gratefully thank the bwGRiD project [1] for the computational resources. Also we want to thank the reviewers for helpful comments.

References

[1] bwgrid. member of the german d-grid initiative, funded by the ministry for education and research and the ministry for science, research and arts baden-wuerttemberg. `http://www.bw-grid.de`.

[2] J. Argelich, C. M. Li, F. Manyà, and J. Planes. Max-sat 2009 fourth max-sat evaluation. `http://www.maxsat.udl.es/09/`.

[3] S. Darras, G. Dequen, L. Devendeville, and C. M. Li. On inconsistent clause-subsets for max-sat solving. In C. Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 225–240. Springer, 2007.

[4] J. Larrosa, F. Heras, and S. de Givry. A logical approach to efficient max-sat solving. *Artif. Intell.*, 172(2-3):204–233, 2008.

[5] C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. Exploiting cycle structures in max-sat. In O. Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 467–480. Springer, 2009.

[6] C. M. Li, F. Manyà, and J. Planes. Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In P. van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 403–414. Springer, 2005.

[7] C.-M. Li, F. Manyà, and J. Planes. Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*, pages 86–91. AAAI Press, 2006.

[8] H. Lin, K. Su, and C. M. Li. Within-problem learning for efficient lower bound computation in max-sat solving. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 351–356. AAAI Press, 2008.

[9] R. Miyashiro and T. Matsui. Semidefinite programming based approaches to the break minimization problem. *Computers & Operations Research*, 33(7):1975–1982, 2006.

[10] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI'99: Proceedings of the 16th international joint conference on Artifical intelligence*, pages 542–547, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[12] H. Xu, R. A. Rutenbar, and K. Sakallah. sub-sat: a formulation for relaxed boolean satisfiability with applications in routing. In *ISPD '02: Proceedings of the 2002 international symposium on Physical design*, pages 182–187, New York, NY, USA, 2002. ACM.

[13] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.