Solving MAXSAT by Solving a Sequence of Simpler SAT Instances

Jessica Davies and Fahiem Bacchus

Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada, M5S 3H5
{jdavies,fbacchus}@cs.toronto.edu

Abstract. MAXSAT is an optimization version of Satisfiability aimed at finding a truth assignment that maximizes the satisfaction of the theory. The technique of solving a sequence of SAT decision problems has been quite successful for solving larger, more industrially focused MAXSAT instances, particularly when only a small number of clauses need to be falsified. The SAT decision problems, however, become more and more complicated as the minimal number of clauses that must be falsified increases. This can significantly degrade the performance of the approach. This technique also has more difficulty with the important generalization where each clause is given a weight: the weights generate SAT decision problems that are harder for SAT solvers to solve. In this paper we introduce a new MAXSAT algorithm that avoids these problems. Our algorithm also solves a sequence of SAT instances. However, these SAT instances are always simplifications of the initial MAXSAT formula, and thus are relatively easy for modern SAT solvers. This is accomplished by moving all of the arithmetic reasoning into a separate hitting set problem which can then be solved with techniques better suited to numeric reasoning, e.g., techniques from mathematical programming. As a result the performance of our algorithm is unaffected by the addition of clause weights. Our algorithm can, however, require solving more SAT instances than previous approaches. Nevertheless, the approach is simpler than previous methods and displays superior performance on some benchmarks.

1 Introduction

MAXSAT is an optimization version of Satisfiability (SAT) that is defined for formulas expressed in Conjunctive Normal Form (CNF). Whereas SAT tries to determine whether or not a satisfying truth assignment exists, MAXSAT tries to find a truth assignment that maximizes the satisfaction of the formula. In particular, if each clause of the CNF formula is given a weight, MAXSAT tries to find a truth assignment that maximizes the sum of the weights of the clauses it satisfies (or equivalently minimizes the weight of the clauses it falsifies).

Various special cases can be defined. With only unit weights, MAXSAT becomes the problem of maximizing the number of satisfied clauses. If some of the clauses must be satisfied (**hard clauses**) they can be given infinite weight, while the other clauses are given unit weight indicating that they can be falsified if

J. Lee (Ed.): CP 2011, LNCS 6876, pp. 225–239, 2011.

necessary (**soft clauses**). In this case we have a Partial MAXSAT problem. If we allow non-unit weights, but no hard clauses, we have a Weighted MAXSAT problem. Finally, with non-unit weights and hard clauses we have a Weighted Partial MAXSAT problem.

In this paper we provide a new approach for solving MAXSAT problems that can be applied to any of these special cases. Our algorithm uses the approach of solving MAXSAT by solving a sequence of SAT tests. Recent international MAXSAT Evaluations have provided empirical evidence that the sequence of SAT tests approach tends to be more effective on the larger more industrially focused problems used in the evaluation. In contrast, the competitive approach of using branch and bound search seems to traverse its search space too slowly to tackle these larger problems effectively.

Previous works employing a sequence of SAT tests have used various techniques to convert the optimization problem into a sequence of decision problems, each of which is then encoded as a SAT problem and solved with a modern SAT solver. Letting W be the sum of the weights of the soft clauses, the typical decision problem used is "are $W - wt$ soft clauses along with all of the hard clauses satisfiable." Typically wt starts off at zero and is increased to the next feasible value every time the answer to the decision problem is no. The solution to the MAXSAT problem is the smallest value of wt for which the decision problem becomes satisfiable. This approach is very successful when only a few decision problems must be posed before a solution is found. However, each SAT decision problem is harder to solve than the previous, and performance can be significantly degraded as more and more decision problems must be solved.

In our approach, on the other hand, we utilize a sequence of SAT problems that become progressively easier. In particular, the SAT solver is only ever asked to solve problems that are composed of a subset of the clauses of the original MAXSAT problem. Our approach moves the arithmetic optimization component of the MAXSAT problem off into a different solver that is more suitable for such reasoning. Modern SAT solvers are based on resolution, and hence can have difficulties with inferences that require counting and other arithmetic reasoning. By separating the two components of satisfiability and optimization present in MAXSAT problems our approach can more effectively utilize the strengths of a SAT solver as well as exploiting the strengths of other solvers, like integer programming solvers, that are known to provide powerful arithmetic reasoning.

In the rest of the paper we first present some necessary background. After this we prove a simple theorem from which we obtain our new algorithm, prove its correctness, and then provide some further insights which allow us to improve our algorithm. The algorithm we present is very simple, but there are some issues that arise when implementing it. We discuss some of these next, followed by a discussion of the most closely related work. We then present various empirical results demonstrating that our approach is viable, and finally we close with some conclusions.

2 Background

A propositional formula in CNF is a conjunction of clauses, each of which is a disjunction of literals, each of which is a propositional variable or the negation of a propositional variable. Given a CNF formula a truth assignment is an assignment of *true* or *false* to all of the propositional variables in the formula.

A MAXSAT problem is specified by a CNF formula $\mathcal{F}$ along with a real valued weight for every clause in the formula (previous works have often required the weights to be integer but we do not require such restrictions in our approach). Let $wt(c)$ denote the weight of clause c. We require that $wt(c) > 0$ for every clause. (Clauses with weight zero can be removed from $\mathcal{F}$ without impact).

Some clauses might be hard clauses, indicated by them having infinite weight. Clauses with finite weight are called soft clauses. We use $hard(\mathcal{F})$ to indicate the hard clauses of $\mathcal{F}$ and $soft(\mathcal{F})$ the soft clauses. Note that $\mathcal{F} = hard(\mathcal{F}) \cup soft(\mathcal{F})$.

We define the function *cost* as follows: (a) if H is a set of clauses then $cost(H)$ is the sum of the clause weights in H ($cost(H) = \sum_{c \in H} wt(c)$); and (b) if π is a truth assignment to the variables in $\mathcal{F}$ then $cost(\pi)$ is the sum of the weights of the clauses falsified by π ($\sum_{\{c \mid \pi \not\models c\}} wt(c)$).

A solution to $\mathcal{F}$ is a truth assignment π to the variables of $\mathcal{F}$ with minimum cost. (Equivalently π maximizes the sum of the weights of the satisfied clauses). We let $mincost(\mathcal{F})$ denote the cost of a solution to $\mathcal{F}$.

For simplicity, in our formal results we will assume that $hard(\mathcal{F})$ is satisfiable and that $\mathcal{F}$ is unsatisfiable. It is straightforward to extend our formal results to deal with these corner cases, but doing so is a distraction from the core ideas. Furthermore, from a practical point of view both conditions can be easily tested with a SAT solver and if either is violated we immediately know $mincost(\mathcal{F})$: if $hard(\mathcal{F})$ is unsatisfiable then $mincost(\mathcal{F}) = \infty$ and any truth assignment is a "solution"; and if $\mathcal{F}$ is satisfiable then $mincost(\mathcal{F}) = 0$ and the SAT solution is also an MAXSAT solution.

A **core** κ for a MAXSAT formula $\mathcal{F}$ is a subset of $soft(\mathcal{F})$ such that $\kappa \cup hard(\mathcal{F})$ is unsatisfiable. That is, all truth assignments falsify at least one clause of $\kappa \cup hard(\mathcal{F})$. Cores can be fairly easily extracted from modern SAT solvers.

Given a set of cores $\mathcal{K}$ a **hitting set**, hs, of $\mathcal{K}$ is a set of soft clauses such that for all $\kappa \in \mathcal{K}$ we have that $hs \cap \kappa \neq \emptyset$. Since every core κ is a set of soft clauses it is not restrictive to also force hs to be a set of soft clauses. We say that hs is a **minimum cost hitting set** of $\mathcal{K}$ if it is (a) a hitting set and (b) $cost(hs) \leq cost(H)$ for every other hitting set H of $\mathcal{K}$.

There have been two main approaches to building MAXSAT solvers. The first approach is to utilize the logical structure of the CNF input to enable the computation of lower-bounds during a branch and bound search, e.g., [7,11]. The second approach is to reduce the problem to solving a sequence of SAT problems. In previous work (see Section 4) these SAT problems typically encode the decision problem: "is $mincost(\mathcal{F}) = k$." Starting with $k = 0$, when the answer from the SAT solver is no (i.e., the formula is unsatisfiable), the next lowest possible value k^+ for k is computed from the core returned by the SAT solver. The next SAT problem then encodes the decision problem "is $mincost(\mathcal{F}) = k^+$".

Recent MAXSAT Evaluations [3] have indicated that these two approaches have different coverage. That is, on some problems the branch and bound approach is significantly better, while on other problems the sequence of SAT problems approach is significantly better. In previous work we had investigated using clause learning to improve the lower bounds computed by a branch and bound solver [5]. In working to improve the performance of this lower bounding technique, related ideas were uncovered that lead to a new approach to solving MAXSAT using a sequence of SAT problems. Since this approach was likely to solve a different set of problems than our branch and bound solver we implemented these ideas in a new solver. This paper reports on our new approach.

3 Solving Maxsat with Simpler SAT Instances

The approach we present in this paper involves solving MAXSAT by solving a sequence of SAT problems. In contrast to prior approaches, however, the SAT problems we need to solve become simpler rather than more complex. In particular, the various encodings of the decision problem $mincost(\mathcal{F}) = k$ that have been used in previous work involve an increasing amount of arithmetic reasoning or involve increasing the size of the theory. For example, in the recent approach of [2] the decision problems contain an increasing number of pseudo-boolean constraints (linear constraints over boolean variables). Counting and arithmetic reasoning is often difficult for a SAT solver since such solvers are based on resolution. There are a number of known examples, e.g., the Pigeon Hole Principle, where resolution requires an exponential number of steps to reach a conclusion that can be quickly deduced by, e.g., reasoning directly with linear equations.

In our approach we split the problem into two parts. In one part we compute minimum cost hitting sets, while in the other part we test the satisfiability of *subsets* of the original problem. In this way we move the arithmetic reasoning into the hitting set solver, allowing the SAT solver to deal with only the logical/satisfiability structure of the original problem. Furthermore, the sequence of satisfiability problems that have to be solved can only become easier. However, the hitting set computations can and do become harder. Our thesis is that by splitting the problem in this manner we can more effectively exploit both the strengths of modern SAT solvers as well the strengths of solvers that are effective at performing the arithmetic reasoning required, e.g., integer programming solvers like CPLEX. Our empirical results provide some evidence in support of our thesis, but also indicate that there is a rich design space in exactly how best to perform this split between satisfiability testing and hitting set computations that remains to be more fully explored.

Our approach is based on a simple theorem.

Theorem 1. *If $\mathcal{K}$ is a set of cores for the* MAXSAT *problem $\mathcal{F}$, hs is a minimum cost hitting set of $\mathcal{K}$, and π is a truth assignment satisfying $\mathcal{F} - hs$ then $mincost(\mathcal{F}) = cost(\pi) = cost(hs)$.*

Proof: $mincost(\mathcal{F}) \leq cost(\pi)$ as $mincost(\mathcal{F})$ is the minimum over all possible truth assignments. $cost(\pi) \leq cost(hs)$ as the clauses π falsifies are a subset of hs

Algorithm 1. Algorithm 1 for Solving MAXSAT

```
1 MAXSAT-solver-1 (F)
2 K = ∅
3 while true do
4 |   hs = FindMinCostHittingSet(K)
5 |   (sat?,κ) = SatSolver(F − hs)
  |   ; // If SAT, κ contains the satisfying truth assignment.
  |   ; // If UNSAT, κ contains an UNSAT core.
6 |   if sat? then
7 |   |   break ; // Exit While Loop
  |   // Add new core to set of cores
8 |   K = K ∪ {κ}
9 return (κ, cost(κ))
```

(π satisfies all clauses in $\mathcal{F}-hs$). On the other hand $mincost(\mathcal{F}) \geq cost(hs)$. Any truth assignment must falsify at least one clause from every core $\kappa \in \mathcal{K}$. Thus for any truth assignment τ, $cost(\tau)$ must include at least the cost of a hitting set of $\mathcal{K}$. This cannot be any less than $cost(hs)$ which has minimum cost. ■

Theorem 1 immediately yields the simple algorithm for solving MAXSAT shown as Algorithm 1. The algorithm starts off with an empty set of cores $\mathcal{K}$. At each stage it computes a minimum cost hitting set hs via the function "FindMinCostHittingSet" and calls a SAT solver to determine if $\mathcal{F} - hs$ is satisfiable. If it is the SAT solver returns (*true*, κ) with κ set to a satisfying assignment for $\mathcal{F}-hs$, otherwise the SAT solver returns (*false*, κ) with κ set to a core of $\mathcal{F}-hs$. New cores are added to $\mathcal{K}$, while satisfying assignments cause the algorithm to terminate.

Observation 1. *Algorithm 1 correctly returns a solution to the inputted* MAXSAT *problem $\mathcal{F}$. That is, it returns a truth assignment κ for $\mathcal{F}$ that achieves $mincost(\mathcal{F})$.*

Proof: First we observe that Algorithm 1 only returns when it breaks out of the while loop, and this occurs only when the current $\mathcal{F}-hs$ is satisfiable. Since in this case hs is a minimum cost hitting set of a set of cores and κ is a truth assignment satisfying $\mathcal{F}-hs$, we have by Theorem 1 that $cost(\kappa) = mincost(\mathcal{F})$. This shows that the algorithm is sound.

Second, to show that the algorithm is complete we simply need to observe that it must terminate. Notice, that since $\mathcal{F}$ is a finite set of clauses, the set of cores of $\mathcal{F}$ must also be finite. Each iteration of the while loop computes a new core of $\mathcal{F}$ and adds it to $\mathcal{K}$. This core cannot be the same as any previous core, hence the while loop must eventually terminate. Consider the hitting set hs computed at line 4 prior to the computation of κ at line 5. $\kappa \cap hs = \emptyset$ since $\kappa \subseteq (\mathcal{F} - hs)$. However, for any previously computed core κ^- we have that $\kappa^- \cap hs \neq \emptyset$ since hs is a hitting set of all previous cores. Hence for all previous cores κ^- we have that $\kappa \neq \kappa^-$. ■

3.1 Realizable Hitting Sets

In this section we show that the hitting sets considered by Algorithm 1 can be further constrained. This can benefit both the time spent calculating the hitting sets, and the overall number of iterations or SAT solving episodes.

Definition 1. *A hitting set H (i.e., a set of clauses) is* ***realizable*** *in a* MAXSAT *problem $\mathcal{F}$ if there exists a truth assignment τ such that (a) for each clause $c \in H$, $\tau \not\models c$, and (b) $\tau \models hard(\mathcal{F})$. Otherwise H is said to be* ***unrealizable****.*

An example of an unrealizable hitting set is one that contains clauses c_1, c_2 with a variable $x \in c_1$ and $\neg x \in c_2$, since all truth assignments satisfy either c_1 or c_2. Next, we show that Algorithm 1 does not gain anything by encountering such unrealizable minimum hitting sets.

Corollary 1 (Of Theorem 1). *Let $\mathcal{K}$ be a set of cores of $\mathcal{F}$ and hs be a minimum cost hitting set of $\mathcal{K}$. If hs is unrealizable, then $\mathcal{F} - hs$ is unsatisfiable.*

Proof: For contradiction, suppose $\pi \models \mathcal{F} - hs$. Then $\pi \models hard(\mathcal{F})$ and since hs is unrealizable, π satisfies some clause in hs. So F_π the set of clauses falsified by π (a) is a strict subset of hs and (b) is a hitting set of $\mathcal{K}$. But then $cost(\mathcal{F}_\pi) < cost(hs)$ which contradicts the fact that hs is a minimum cost hitting set of $\mathcal{K}$. ■

Corollary 1 means that any time line 5 of Algorithm 1 returns an unrealizable hs, at least one more iteration of the while loop will be required. Yet in fact, there might be enough information already in the set of cores $\mathcal{K}$ to terminate right away. To see this, remember the aim in solving MAXSAT is to find a truth assignment of minimum cost. Let π be any truth assignment and let $F_\pi = \{c | \pi \not\models c\}$ be the set of clauses falsified by π. Given a set of cores $\mathcal{K}$ we know that π must falsify at least one clause from each core in $\mathcal{K}$. This means that we can partition F_π into two sets, hs_π a hitting set of $\mathcal{K}$, and $F_\pi - hs_\pi$ the remaining falsified clauses. This also partitions the cost of π into two components, $cost(\pi) = cost(hs_\pi) + cost(F_\pi - hs_\pi)$.

Theorem 1 says that if $cost(hs_\pi)$ is minimum (less than or equal to the cost of any hitting set of $\mathcal{K}$), and $cost(F_\pi - hs_\pi)$ is zero (i.e., $F_\pi - hs_\pi = \emptyset$), then π must be a minimum cost truth assignment as no other truth assignment can achieve a lower cost. Looking more closely, however, we can see that the first condition is more stringent than necessary. We do not need hs_π to be a minimum cost hitting set of $\mathcal{K}$, we only need that $cost(hs_\pi) \leq cost(hs_\tau)$ for all other truth assignments τ. We will then have that $cost(\pi) = cost(hs_\pi) + 0 \leq cost(hs_\tau) + cost(\mathcal{F}_\tau - hs_\tau) = cost(\tau)$ for all other truth assignments τ. That is, π will be a solution. Going even further we see that we do not need to consider all truth assignments τ. If τ falsifies a hard clause of $\mathcal{F}$ it will immediately have cost ∞, and thus will necessarily be at least as expensive as π.

Realizable hitting sets are relevant because the minimum cost hitting set of $\mathcal{K}$ might not be realizable. In particular, given the current set of cores $\mathcal{K}$ in Algorithm 1, there might be some truth assignment π which satisfies $\mathcal{F} - hs_\pi$

Algorithm 2. Algorithm 2 for Solving MAXSAT

```
1 MAXSAT-solver-2 (F, COND)
  ; // COND must be satisfied by all hitting sets realizable in F.
  Identical to Algorithm 1 except we replace
  hs = FindMinCostHittingSet(K)
  by
  hs = FindMinCostHittingSetSatisfyingCondition(K,COND)
```

(i.e., $F_\pi - hs_\pi = \emptyset$) and for which $cost(hs_\pi) \leq cost(hs_\tau)$ for any other truth assignment τ where $\tau \models hard(\mathcal{F})$. This means that (a) hs_π is a minimum cost realizable hitting set of $\mathcal{K}$, (b) if we pass $\mathcal{F} - hs_\pi$ to the SAT solver it will return π (or some equally good truth assignment) as a satisfying assignment, and (c) we have solved $\mathcal{F}$.

However, hs_π might not be a minimum cost hitting set of $\mathcal{K}$. There might be another hitting set hs that has minimum cost that is lower than the cost of hs_π, but is unrealizable. In Algorithm 1, hs would be selected and the SAT solver invoked with $\mathcal{F} - hs$. This will necessarily cause another core to be returned, and Algorithm 1 will then have to go through another iteration.

Corollary 1 indicates that we can improve on Theorem 1 and Algorithm 1 by computing minimum cost *realizable* hitting sets rather than unconstrained minimum cost hitting sets. Realizability requires a SAT test so it can be expensive. Hence, we improve Theorem 1 and Algorithm 1 in a more general way. In particular, we can search for a minimum cost hitting set that satisfies any condition that is satisfied by all realizable hitting sets. For example, realizability is one such condition. A simpler condition that is easy to test is to ensure that no clauses in the hitting set contain conflicting literals: this condition is also satisfied by all realizable hitting sets.

Theorem 2. *If $\mathcal{K}$ is a set of cores for the* MAXSAT *problem $\mathcal{F}$, COND is a condition satisfied by all hitting sets that are realizable in $\mathcal{F}$, hs is a hitting set of $\mathcal{K}$ that satisfies COND and has minimum cost among all hitting sets of $\mathcal{K}$ satisfying COND, and π is a truth assignment satisfying $\mathcal{F} - hs$ then $mincost(\mathcal{F}) = cost(\pi) = cost(hs)$.*

Proof: $mincost(\mathcal{F}) \leq cost(\pi) \leq cost(hs)$ by exactly the same argument as for Theorem 1. Furthermore $mincost(\mathcal{F}) \geq cost(hs)$. Any truth assignment that satisfies $hard(\mathcal{F})$ must falsify a hitting set of $\mathcal{K}$ that satisfies *COND*. Thus for any truth assignment τ, $cost(\tau)$ must include at least the cost of a hitting set of $\mathcal{K}$ that is at least as great as $cost(hs)$: $cost(hs)$ is minimum among all hitting sets of $\mathcal{K}$ satisfying *COND*. ■

The improved version of Algorithm 1 is shown as Algorithm 2. Algorithm 2 takes as input a condition satisfied by all hitting sets that are realizable in $\mathcal{F}$. It now searches for a minimum cost hitting set that satisfies this condition. This can potentially cut down the number of iterations of the **while** loop, reducing the number of cores that have to be generated.

Observation 2. *Algorithm 2 correctly returns a solution to the inputted* MAXSAT *problem* $\mathcal{F}$.

Proof: The proof that Algorithm 1 is correct applies using Theorem 2 in place of Theorem 1. ■

Finally, we close this section with a brief comment about complexity. The worst case complexity of solving MAXSAT with a branch and bound solver is $2^{O(n)}$ where n is the number of variables. However, the worst case complexity of our algorithm is worse. There are $2^{O(m)}$ possible cores where m is the number of clauses. This provides a worst case bound on the number of iterations executed in the algorithm. Each iteration requires solving a SAT problem of $2^{O(n)}$ and a hitting set problem of $2^{O(m)}$ (one has to examine sets of clauses to find a hitting set). This leaves us with $2^{O(m)} \times 2^{O(m)} = 2^{O(m)}$ as the worst case complexity. Typically the number of clauses m is much larger than the number of variables n.

However, from a practical point of view we only expect our algorithm to work well when the number of cores it has to compute is fairly small. The empirical question is whether or not this tends to occur on problems that arise in various applications.

3.2 Implementation Techniques

There are two issues to be addressed in implementing our algorithm. First is the use of a SAT solver to compute new cores, and second is the computation of minimum cost hitting sets.

Extracting Cores. We use MiniSat-2.0 to compute cores and satisfying assignments. There is a simple trick that can be employed in MiniSat to make extracting cores easy. Following previous work we add a unique "relaxation variable" to each clause of $soft(\mathcal{F})$. So soft clause c_i becomes $c_i \cup \{b_i\}$ where b_i appears nowhere else in the new theory. The hard clauses of $\mathcal{F}$ are unchanged. If b_i is set to *true*, c_i becomes true and imposes no further constraints on the theory. If b_i is set to *false*, c_i is returned to its original state. To solve $\mathcal{F} - hs$ we set the b variables associated with the clauses in hs to *true*, and all other b variables to *false*. These b variable assignments are added as "assumptions" in MiniSat. MiniSat then solves the remaining problem $\mathcal{F} - hs$ and if this is UNSAT it computes a conflict clause over the assumptions—the set of assumptions that lead to failure. The *true* b variables do not impose any constraints so they cannot appear in the conflict clause. Instead, the conflict claused contains the set of *false* b variables that caused UNSAT. The core is simply the set of clauses associated with the b variables of the computed conflict.

An important factor in the performance of our algorithm is the diversity of the cores returned by the SAT solver. In the first phase, we compute as many disjoint cores as possible. The hitting set problems for disjoint cores are easy, and the cost of the minimum cost hitting set increases at each iteration. Typically, however, it is necessary to continue beyond this disjoint phase. Nevertheless we want the

SAT solver to return a core that is as different as possible from the previous cores. To encourage this to happen we employ the following two techniques in the SAT solver. (1) Although it can be shown to be sound to retain learnt clauses and reuse them in subsequent SAT solving calls, we found that doing so reduces the diversity of the returned core. Hence we removed all previously learnt clauses at the start of each SAT call. (2) We inverted the VSIDS scores that were computed during the previous SAT call. The VSIDS score makes the SAT solver branch on variables appearing most frequently in the learnt clauses of the previous SAT call. By inverting these scores the SAT solver tends to explore a different part of the space and tends to find a more diverse new core. Finally, it is also useful to obtain cores that are as small as possible (such cores are more constraining so they make the hitting set problem easier to solve). So after computing a core κ we feed it back into the SAT solver to see if a subset of κ can be detected to be UNSAT. We continue to do this until κ cannot be minimized any further.

Computing a minimal cost hitting set. We employed two different techniques for computing minimal cost hitting sets. The first technique is to encode the problem as an integer linear program (ILP) and invoke an ILP solver to solve it. In our case we utilized the CPLEX solver. The minimal cost hitting set problem is the same as the minimum cost set cover problem and standard ILP encodings exist, e.g., [13]. We used the encoding previously given in [5]. Briefly, for each clause c_i appearing in a core there is a 0/1 variable x_i; for each core there is the constraint that the sum over the x_i variables of the clauses it contains is greater or equal to 1; and the objective is to minimize the sum of $wt(c_i) \times x_i$. Using CPLEX worked well, but it is not clear how to solve for minimal cost realizable hitting sets—to do so would seem to require adding the satisfiability constraints of the hard clauses to the ILP model, and it is well known that ILP solvers are not very effective at dealing with these highly disjunctive constraints.

The second approach we used was our own branch and bound hitting set solver. We utilized a dancing links representation of the hitting set problem [9], and at each node branched on whether or not a clause was to be included or excluded from the hitting set. The main advantage of the dancing links representation is that it allowed us to simplify the representation after each decision. We performed two types of simplification. First, we simplified the representation to account for the decision made (e.g., if we decide to include a clause c_i we could remove all cores that c_i hit from the remaining hitting set problem). These simplifications are well described by Knuth in [9]. Second, we use the simplifications provided in [14] to further reduce the remaining problem. These latter simplifications involve two rules (a) if a core κ_1 has now become a subset of another core κ_2 we know that in hitting κ_1 we must also hit κ_2 so κ_2 can be removed; and (b) if a clause c_1 now appears in a subset of the cores that another clause c_2 appears in and $wt(c_1) \geq wt(c_2)$ we know that we can replace c_1 with c_2 in any hitting set so c_1 can be removed. These simplifications take time but overall in our implementation we found that they yielded a net improvement in solving times.

We additionally experimented with various lower bounds in the hitting set solver. In particular, we tried both of the simple to compute lower bounds given in [5]. Eventually, however, we found that a linear programming relaxation, although more expensive, yielded sufficiently superior bounds so as to improve the overall solving times. This LP relaxation was simply the current reduced hitting set problem encoded using the ILP encoding specified above with the integrality constraints removed. We used CPLEX to solve the LP.

We found that our branch and bound solver did not solve the hitting set problem as efficiently as CPLEX with the ILP encoding. However, with it we were able to implement the realizability condition forcing the solver to find a minimum cost realizable hitting set. This was accomplished by making additional calls to a SAT solver. At each node of the search tree, we performed the following test. If H was the set of clauses currently selected by the branch and bound solver to be in the hitting set, then we applied unit propagation to the theory containing all of the hard clauses of $\mathcal{F}$ along with the negation of every literal in every clause in H. If unit propagation revealed an inconsistency, we backtracked from the node since it could not lead to a realizable hitting set. Whenever branch and bound found a better-cost hitting set, we used the complete SAT test to check if it was realizable. Enforcing realizability also forced us to turn off the second simplification rule given above: removing a clause c_i because it is subsumed by another clause c_j is no longer valid as c_i rather than c_j might be needed for the hitting set to be realizable.

With the addition of realizability we found that our branch and bound hitting set solver was much more competitive with CPLEX on some problems. There are still a number of other improvements to our branch and bound that remain to be tested, including OR-Decomposition [8], caching, and alternate lower bounding techniques like Lagrangian relaxation [15].

4 Related Work

The main prior works utilizing a sequence of SAT tests to solve MAXSAT began with the work of Fu and Malik [6], and include SAT4J [4], WPM1, PM2, and WPM2 [1,2], and Msuncore [12]. As mentioned above there has also been work on branch and bound based solvers but such solvers are not directly comparable with the sequence of SAT solvers: each type of solver is best suited for different types of problems.

All of the sequence of SAT solvers utilize relaxation variables added to the soft clauses of $\mathcal{F}$ as described in Section 3.2, along with arithmetic constraints on which of these relaxation variables can true. Let $\mathit{soft}(\mathcal{F}) = \{c_1, \ldots, c_k\}$ and the corresponding relaxation variables be $\{b_1, \ldots, b_k\}$. SAT4J adds to $\mathcal{F}$ the constraint $\sum_{i=1}^{i=k} \mathit{wt}(c_i)b_i < \mathit{UB}$ encoded in CNF where UB is the current upper bound on $\mathit{mincost}(\mathcal{F})$. If this theory (with the numeric constraint encoded into SAT) is satisfiable UB is decreased and satisfiability retested until the theory transitions from SAT to UNSAT.

The other algorithms, like our approach, work upwards from UNSAT to SAT. And like SAT4J they add arithmetic constraints on the b variables as more

cores are discovered. PM2 works only with unweighted clauses. At each iteration that produces a core, PM2 increments the upper bound on the total number of b variables that can be true. PM2 also uses the cores to derive *lower bounds* on different subsets of b variables. In the case of WPM2, each SAT test that returns UNSAT yields a core. This core is widened to include all previous cores it intersected with, and then an arithmetic constraint is added saying that the sum of the b variables in the widened core must have an increased weight of true b variables. Simultaneously another arithmetic constraint is added placing an upper bound on the weight of true b variables in the widened core. These constraints are formulated in such a manner that when the theory transitions from UNSAT to SAT, $mincost(\mathcal{F})$ has been computed.

The arithmetic constraints used in WPM1 and Msuncore are simpler. However, the theory is becoming more complex as the approach involves duplicating clauses. In particular, all of the clauses of the discovered core are duplicated. One copy gets a new b variable and a clause weight equal to the minimum weight clause of the core, while the other copy has the same weight subtracted from it. Finally, a new constraint is added to make the new b variables sum to one.

In contrast to these approaches the approach we present here involves a sequence of simpler SAT problems. There are no arithmetic constraints added to the SAT problem and no clauses are duplicated. Instead, the arithmetic constraints specifying that at least one clause from every core needs to be falsified are dealt with directly by the minimum hitting set solver. In addition, none of the previous approaches have looked at the issue of making sure that the relaxed clauses (i.e., the clauses with turned on b variables) are realizable.

Another closely related work is [5]. Although this work was focused on a branch and bound method, it also utilized the deep connection between hitting sets and MAXSAT solutions that we were able to further exploit here.

5 Empirical Results

We investigated the performance of our proposed algorithms on a variety of industrial and crafted instances, covering all weight categories: unweighted (MS), partial (PMS), weighted (WMS) and weighted partial (WPMS) MAXSAT. Our results suggest that our approach can solve 17 problems that have not been solved before, and can reasonably handle a variety of MAXSAT problems. We also present results on problems with diverse weights, to further illustrate the advantages of our approach.

We ran experiments with all available MAXSAT solvers that use a sequence of SAT problem approach: Msuncore, WBO [12], PM2, WPM1, WPM2 [1,2], SAT4J [4], and Qmaxsat [10].

In order to evaluate the effectiveness of our approach on industrial instances, we ran tests on all 1034 unsatisfiable Industrial instances from the 2009 MAXSAT Evaluation [3], as well as the 116 unsatisfiable WMS instances from the Crafted category. All experiments were conducted with a 1200 second timeout and 2.5GB memory limit, on 2.6GHz AMD Opteron 2535 processors.

In Table 1, we report the number of instances solved and the total runtime on solved instances, by benchmark family. Results are shown for SAT4J, WPM1 and WPM2, since these three solvers represent all existing algorithms that use a sequence of SAT approach and can handle weighted clauses (WPM1 solved more instances overall than Msuncore and WBO). The last two columns show our results for a version of our solver that implements Algorithm 1 and uses CPLEX to solve the ILP formulation of the hitting set problem. Although we don't solve the most problems overall, the families where we do perform best are highlighted in bold. In general, Algorithm 2 solved fewer problems than Algorithm 1 so its results are omitted from this table.

However, there were four benchmark families in which enforcing the realizability condition paid off. In particular, Algorithm 2 solved 44 instances that Algorithm 1 could not solve. These instances are shown in Table 2, which lists the number of instances solved, their average optimum, the average number of iterations Algorithm 1 performed before the timeout, and the number of iterations and runtime for Algorithm 2. We observe that the number of iterations that Algorithm 2 requires to solve the problem is usually significantly fewer than Algorithm 1 performs. This demonstrates that constraining the hitting sets to be realizable can reduce the number of iterations, on some problems.

Table 1. The number of instances solved, and total runtime on solved instances for the 2009 MAXSAT Evaluation industrial and WMS crafted instances

		SAT4J		WPM1		WPM2		Alg1:CPLEX	
Family	#	#	Time	#	Time	#	Time	#	Time
ms/CirDeb	9	7	600	9	178	8	1051	9	395
ms/Sean	108	27	2890	86	6952	73	7202	73	6775
pms/bcp-fir	59	10	38	55	1470	48	2379	21	1873
pms/bcp-simp	138	132	415	131	796	137	1272	131	1326
pms/bcp-SU	38	9	591	13	1596	21	5299	19	3287
pms/bcp-msp	148	96	698	24	731	69	3282	5	2085
pms/bcp-mtg	215	199	1280	181	2651	215	172	102	2384
pms/bcp-syn	**74**	**24**	**2851**	**33**	**731**	**34**	**511**	**60**	**2761**
pms/CirTrace	4	4	2013	0	0	4	1193	0	0
pms/HapAsbly	**6**	**0**	**0**	**2**	**771**	**5**	**143**	**5**	**85**
pms/pbo-logenc	128	128	4229	72	5535	72	8799	78	856
pms/pbo-rtg	15	15	3236	15	16	15	131	14	453
pms/PROT	12	3	876	1	22	3	486	1	9
wpms/up-10	**20**	**20**	**71**	**20**	**109**	**20**	**525**	**20**	**62**
wpms/up-20	20	20	74	20	119	20	601	20	78
wpms/up-30	20	20	77	20	124	20	658	20	146
wpms/up-40	20	20	76	20	134	20	731	20	94
wms/KeXu	34	8	3774	1	395	16	1978	10	1342
wms/RAM	15	4	186	2	326	2	208	1	2
wms/CUT-DIM	62	2	0	4	0	3	0	4	847
wms/CUT-SPIN	5	1	7	0	0	0	0	1	132
Total	1150	749	23990	709	22665	805	36631	614	25002

Table 2. Results on instances Algorithm 2 can solve within 1200 s but Algorithm 1 cannot

		Avg	Alg1:CPLEX	Alg2:B&B	
Family	#	OPT	Iter	Iter	Time
ms/Sean	4	1	13	67	434
pms/bcp-msp	26	99	460	121	204
pms/bcp-mtg	13	8	2198	757	258
pms/bcp-syn	1	6	80	53	295

Table 3. Detailed results on newly solved instances, from the industrial PMS bcp-syn family. '-' in the Time columns indicates timeout.

		Alg1:CPLEX					Alg1:B&B				Alg2:B&B			
Instance	OPT	Iter	|Core|	MxN	HS	Time	Iter	Nodes	HS	Time	Iter	Nodes	HS	Time
saucier.r	6	80	2885	40x2167	15	-	3		1195	-	53	5	5	295
1_1_10_15	10	89	12	44x77	0	26	86	10	0	40	85	11	0	39
1_1_10_10	12	95	10	47x75	0	44	93	10	0	78	93	12	1	117
300_10_20	17	96	14	48x147	1	148	94	14	1	142	97	17	1	152
300_10_14	19	93	11	46x130	0	35	89	12	0	46	88	16	0	37
300_10_15	19	99	12	49x144	0	62	95	13	1	106	96	17	1	134
300_10_10	21	95	9	47x119	0	13	95	14	0	47	95	18	0	42
ex5.r	37	285	28	132x294	0	116	281	24	4	1187	260	46	5	-
ex5.pi	65	304	25	137x267	0	72	301	23	3	924	271	50	2	511
pdc.r	94	413	10	176x212	0	14	408	19	0	99	389	86	1	537
test1.r	110	278	6	119x176	0	3	277	9	0	17	271	85	0	67
rot.b	115	626	23	288x453	0	304	363	40	4	-	345	91	4	-
bench1.pi	121	330	8	149x290	0	25	331	28	1	298	328	113	1	264
max1024.r	245	747	5	323x377	0	153	734	28	1	1016	635	179	2	-
max1024.pi	259	724	5	310x358	0	200	720	25	2	-	663	187	2	-
prom2.r	278	935	6	385x498	0	61	968	21	0	717	733	225	2	-
prom2.pi	287	914	6	372x484	0	40	966	26	1	846	747	249	2	-
Average	100	364	180	159x368	1	82	347	18	71	397	308	83	2	200

In Table 3 we present more detailed results on 17 instances amoung those in Table 1. These were selected based on the fact that none of the competing solvers were able to solve them, and furthermore, they weren't solved by any other solver in the 2009 and 2010 MAXSAT Evaluations. We report results of using Algorithms 1 and 2 with our B&B solver for the hitting set problem, as well as Algorithm 1 with CPLEX for the hitting set. For each version of our solver, and each instance, we list the number of iterations (i.e. SAT episodes), the average time to solve the hitting set problems (columns 'HS'), and the total runtime. We also report some information about the size of the hitting set problems encountered. Column '|Core|' reports the average number of clauses in the cores. Column 'MxN' reports the average dimensions of the hitting set problem given to CPLEX after the simplification rules have been applied. The 'Nodes' columns give the average number of nodes searched by B&B while solving the hitting set problems. The time the SAT solver takes to generate each core is always

Table 4. The number of iterations and runtimes (s) for an industrial WPMS instance as the number k of distinct weights is increased. '-' indicates failure to solve within 1200 s.

		SAT4J		WPM1		WPM2		WBO		Alg1:CPLEX	
k	Opt	Iter	Time	Iter	Time	Iter	Time	Iter	Time	Iter	Time
1	101	1	3	101	2	102	13	80	4	199	4
2	149	1	-	112	3	133	16	80	4	191	3
4	250	1	-	122	3	140	17	79	4	227	3
8	441	1	-	128	4	163	20	80	2	190	2
10	554	2	-	129	4	175	24	80	4	193	3
16	750	1	-	125	5	3751	-	80	3	161	2
32	1621	3	-	127	7	3066	-	79	2	218	4
64	2857	1	-	126	8	3104	-	80	4	172	3
100	4667	1	-	122	16	95	-	80	4	247	5
128	5994	1	-	126	17	3900	-	79	3	192	2
1000	47187	1	-	128	125	1644	-	80	2	193	3
10000	480011	4	763	127	1069	3074	-	80	3	182	2
100000	5057882	1	-	47	-	3108	-	80	4	184	3

less than 0.02s, so this is not included in the table. Our algorithms seem to be particularly suited to these problems. All three versions of our solver do well, whereas all previous MaxSAT methods fail. This is somewhat surprising since all of the instances are of the PMS type, with no clause weights. However, most of these instances have quite large optimums and therefore require many clauses to be relaxed. This is challenging for prior sequence of SAT approaches, even though the cores of the original MaxSAT theory are quite small. Our approach is able to succeed on these instances because it better exploits the SAT solver's ability to very quickly generate many cores of the original theory.

Sequence of SAT solvers are well suited to industrial problems, which are very large but easily refuted by existing SAT solvers. However, their performance can be adversely affected by the distribution of weights on the soft clauses. At the moment, most of the industrial WPMS benchmark problems have a very small number of distinct weight values. Many real-world applications will require a greater diversity of weights.

In order to investigate our solver's performance on problems with diverse weights, we created a new set of WPMS instances by increasing the diversity of weights on an existing benchmark instance. We selected an industrial problem that is easy for sequence of SAT solvers, the Linux Upgradeability family in the WPMS Industrial category of the 2009 Max-SAT Evaluation. Note that all instances in this family already have the same underlying CNF, just different weights. We generated 13 new instances for an increasing number $k \in \{2^0, 2^1, ..., 2^6\} \cup \{10^1, 10^2, ..., 10^6\}$ of distinct weights. Given k, the weight for each soft clause was randomly chosen (with replacement) from the set $\{1, 2, ..., k\}$. The number of iterations and the runtime on each instance is shown in Table 4. We see that WBO and our solver are immune to this type of weight diversification. Their number of SAT solving episodes and their runtimes remain steady as the number of distinct

weights is increased. Although the number of iterations required by WPM1 also doesn't increase with k, the runtimes do increase. We observe that both the number of iterations required, and the runtimes increase significantly for WPM2. SAT4J also has difficulty with these problems.

6 Conclusion

We have presented a new approach to solving MAXSAT via a sequence of SAT problems. We proposed to separate the arithmetic reasoning from the satisfiability testing, allowing the sequence of SAT problems to be simpler rather than more difficult as in previous approaches. The new technique is competitive with previous solvers, and is able to solve some problems previous approaches could not solve. It is also a very simple approach that opens the door for many future improvements.

References

1. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial maxsat through satisfiability testing. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT), pp. 427–440 (2009)
2. Ansótegui, C., Bonet, M.L., Levy, J.: A new algorithm for weighted partial maxsat. In: Proceedings of the AAAI National Conference (AAAI), pp. 3–8 (2010)
3. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The First and Second Max-SAT Evaluations. JSAT 4(2-4), 251–278 (2008)
4. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT 7(2-3), 56–59 (2010)
5. Davies, J., Cho, J., Bacchus, F.: Using learnt clauses in maxsat. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 176–190. Springer, Heidelberg (2010)
6. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Theory and Applications of Satisfiability Testing (SAT), pp. 252–265 (2006)
7. Heras, F., Larrosa, J., Oliveras, A.: Minimaxsat: An efficient weighted max-sat solver. Journal of Artificial Intelligence Research (JAIR) 31, 1–32 (2008)
8. Kitching, M., Bacchus, F.: Exploiting decomposition in constraint optimization problems. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 478–492. Springer, Heidelberg (2008)
9. Knuth, D.E.: Dancing links. In: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, pp. 187–214. Palgrave, Oxford (2000)
10. Koshimura, M., Zhang, T.: Qmaxsat, `http://sites.google.com/site/qmaxsat`
11. Li, C.M., Manyà, F., Mohamedou, N.O., Planes, J.: Resolution-based lower bounds in maxsat. Constraints 15(4), 456–484 (2010)
12. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for weighted boolean optimization. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT), pp. 495–508 (2009)
13. Vazirani, V.: Approximation Algorithms. Springer, Heidelberg (2001)
14. Weihe, K.: Covering trains by stations or the power of data reduction. In: Proceedings of Algorithms and Experiments (ALEX 1998), pp. 1–8 (1998)
15. Wolsey, L.A.: Integer Programming. Wiley, Chichester (1998)