Postponing Optimization to Speed Up MAXSAT Solving

Jessica Davies[1] and Fahiem Bacchus[2]

[1] MIAT, UR 875, INRA, F-31326 Castanet-Tolosan, France
jessica.davies@toulouse.inra.fr
[2] Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada, M5S 3H5
fbacchus@cs.toronto.edu

Abstract. MAXSAT is an optimization version of SAT that can represent a wide variety of important optimization problems. A recent approach for solving MAXSAT is to exploit both a SAT solver and a Mixed Integer Programming (MIP) solver in a hybrid approach. Each solver generates information used by the other solver in a series of iterations that terminates when an optimal solution is found. Empirical results indicate that a bottleneck in this process is the time required by the MIP solver, arising from the large number of times it is invoked. In this paper we present a modified approach that postpones the calls to the MIP solver. This involves substituting non-optimal solutions for the optimal ones computed by the MIP solver, whenever possible. We describe the new approach and some different instantiations of it. We perform an extensive empirical evaluation comparing the performance of the resulting solvers with other state-of-the-art MAXSAT solvers. We show that the best performing versions of our approach advance the state-of-the-art in MAXSAT solving.

1 Introduction

MAXSAT, the optimization version of Satisfiability (SAT), is the problem of finding a minimum cost truth assignment for a set of clauses where a cost is incurred for every falsified clause. It is called MAXSAT since in the simplest case where every clause is equally costly to falsify, a solution will satisfy a maximum number of clauses. In the most general version of MAXSAT some clauses are *hard* incurring an infinite cost if they are falsified, while the other clauses are *soft* incurring some integer cost greater than zero. This most general version of MAXSAT is often called **weighted partial MAXSAT** (WPMS) and is what we address in this paper. Many practical problems can be encoded in MAXSAT, so developing effective ways to solve MAXSAT is an important research topic.

There are two standard methods for solving MAXSAT: using Branch and Bound search (e.g. [9, 14]), and using a sequence of decision problems, usually encoded as SAT (e.g. [2, 3, 5, 15, 10]). In [6] an alternative algorithm for solving MAXSAT, called MAXHS, was presented. MAXHS also solves a sequence of SAT decision problems, but in contrast to existing approaches the SAT problems do not become more difficult for the SAT solver to solve. This is accomplished via a hybrid approach, whereby a SAT solver and a Mixed Integer Linear Program (MIP) solver are used to cooperatively solve the MAXSAT problem using an approach similar to Bender's Decomposition [11]. The MIP solver is used to find optimal solutions which the SAT solver then tests for feasibility. If the solution is not feasible the SAT solver computes a new constraint to add to the MIP model and the MIP solver is invoked again to find a new optimal solution that additionally satisfies the new constraint.

In this paper we investigate a new technique for improving the performance of this hybrid approach. In [7] it has already been shown that the hybrid approach is one of the state-of-the-art approaches to solving MAXSAT, and thus improving this approach is one way of advancing the state-of-the-art.

Analyzing the performance of the hybrid approach indicates that the main bottleneck is the time spent by the MIP solver. This time mostly accumulates from the number of times that the MIP solver must be called: it is called every time the SAT solver computes a new feasibility constraint, in order to derive a new optimal solution satisfying this additional constraint. Although these calls often take relatively little time, after hundreds of separate calls the total time becomes quite significant.

Inspired by an idea presented by Moreno-Centeno and Karp [16] we developed a method for delaying the calls to the MIP solver for as long as possible. We accomplish this by recognizing situations where non-optimal solutions can be used in place of the optimal solutions produced by the MIP solver without impacting the algorithm's correctness. The SAT solver can use these non-optimal solutions to compute its feasibility constraints and the iterations of feasibility and "optimization" can continue. However, since the optimization phase is now an approximation that can be computed cheaply, each iteration is much more efficient. Eventually, however, an optimal solution must be computed to ensure correctness. So our technique postpones, rather than removes, optimization.

We show that our new technique yields a significant improvement in the performance of the hybrid algorithm and makes it the most robust current approach to solving MAXSAT. One of the reasons why we obtain such a good performance improvement is that the MIP solver is not perfectly incremental. By using non-optimal solutions we collect many feasibility constraints before having to compute their optimal solution. This means that each additional call to the MIP solver involves a model that has been augmented by many feasibility constraints, whereas in the previous approach the model was only augmented by a single feasibility constraint. Although the MIP solver can take advantage of its previous computations when called again, it is not perfectly incremental. That is, its solving time when given k new constraints is typically significantly smaller than the sum of its k solving times when it is given these constraints one at a time and asked to compute an optimal solution each time.

The remainder of the paper is organized as follows. Section 2 provides basic definitions. The MAXHS approach is then reviewed in Section 3 and we show that its main bottleneck is the MIP solving time. Section 4 presents the new algorithm, which addresses this issue by allowing the MIP optimization to be postponed. An effective additional enhancement, seeding the MIP model with constraints [7], is described in Section 5. The empirical results are reported in Section 6.

2 Background

A MAXSAT instance is specified by a set of propositional clauses $\mathcal{F}$ each having a positive integer or infinite weight $wt(c)$, $c \in \mathcal{F}$. Clauses with infinite weight are called hard clauses and are collectively denoted by $hard(\mathcal{F})$. The other clauses of $\mathcal{F}$ all have finite weight and are called soft clauses, $soft(\mathcal{F})$ ($\mathcal{F} = hard(\mathcal{F}) \cup soft(\mathcal{F})$).

We define the function *cost* as follows: (a) if H is a set of clauses then $cost(H)$ is the sum of the weights of the clauses in H ($cost(H) = \sum_{c \in H} wt(c)$); and (b) if π is a truth assignment to the variables of $\mathcal{F}$ 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 for $\mathcal{F}$ is a truth assignment π to the variables of $\mathcal{F}$ that satisfies $hard(\mathcal{F})$ and is of minimum cost. We let $mincost(\mathcal{F})$ denote the cost of such a solution. If $hard(\mathcal{F})$ is unsatisfiable, then $\mathcal{F}$ has no solution. In the remainder of the paper, we assume that $hard(\mathcal{F})$ **is satisfiable** (this is easy to test in practice and facilitates clarity). A **core** κ of a MAXSAT formula $\mathcal{F}$ is a subset of $soft(\mathcal{F})$ such that $\kappa \cup hard(\mathcal{F})$ is unsatisfiable. Note that since $hard(\mathcal{F})$ is satisfiable, any solution to $\mathcal{F}$ must falsify at least one clause in κ.

MAXSAT solvers that solve a sequence of decision problems typically insert blocking variables (b-variables) into the soft clauses of the MAXSAT instance.

Definition 1. *If $\mathcal{F}$ is a MAXSAT problem, then its **b-variable relaxation** is a SAT problem $\mathcal{F}^b = \{(c_i \vee b_i) : c_i \in soft(\mathcal{F})\} \cup hard(\mathcal{F})$ where all clause weights are removed. The b-variable b_i appears in the relaxed clause $(c_i \vee b_i)$ and **no where else** in $\mathcal{F}^b$.*

The b-variable relaxation $\mathcal{F}^b$ allows cores of the original MAXSAT formula $\mathcal{F}$ to be computed conveniently, using the **Assumption** mechanism provided by MINISAT [8]. MINISAT can take as input a set of assumptions $\mathcal{A}$, specified as a set of literals, along with a CNF formula $\mathcal{F}$ and then determine if $\mathcal{F} \wedge \mathcal{A}$ is satisfiable. It will return a satisfying truth assignment for $\mathcal{F} \wedge \mathcal{A}$ if one exists. Otherwise it will report unsatisfiability and return a learnt clause c which is a disjunction of negated literals of $\mathcal{A}$. This clause has the property that $\mathcal{F} \models c$. Thus in order to find a core of MAXSAT instance $\mathcal{F}$, we can pass MINISAT the CNF formula $\mathcal{F}^b$ and the set of all negated b-variables as the assumptions. If $\mathcal{F}$ has a core, MINISAT will return UNSAT along with a clause $c = (b_{i_1} \vee \cdots \vee b_{i_k})$ such that $\mathcal{F}^b \models c$ and $\kappa = \{c_{i_1}, ..., c_{i_k}\}$ is a core of $\mathcal{F}$. Any clause over positive b-variables that is entailed by $\mathcal{F}^b$, e.g., $c = (b_{i_1} \vee \cdots \vee b_{i_k})$, is called a **core constraint**.

Besides supporting the computation of cores, the relaxed formula $\mathcal{F}^b$ can also be used to find solutions for the original MAXSAT formula $\mathcal{F}$. To accomplish this we define an objective function $bcost(\pi)$ over truth assignments π to the variables of $\mathcal{F}^b$, equal to the sum of the costs of the clauses whose b-variables are set to *true*: $bcost(\pi) = \sum_{b_i : \pi \models b_i} wt(c_i)$. The minimum *bcost* models of $\mathcal{F}^b$ are MAXSAT solutions.[3]

Proposition 1. *$mincost(\mathcal{F}) = \min_{\pi \models \mathcal{F}^b} bcost(\pi)$. Furthermore, if $\pi \models \mathcal{F}^b$ achieves a minimum value of $bcost(\pi)$, then π restricted to the variables of $\mathcal{F}$ is a solution for $\mathcal{F}$.*

However, $\mathcal{F}^b$ has many models whose *bcost* is greater than necessary. For example, if a model π assigns b_i to *true* even though it also satisfies the soft clause c_i, then the *bcost* of π could be reduced by instead assigning b_i to *false*. We can eliminate such models by modifying $\mathcal{F}^b$ so that the b-variables are forced to be equivalent to the negation of their corresponding soft clauses.

Definition 2. *Let $\mathcal{F}$ be a MAXSAT formula. Then*

$$\mathcal{F}^b_{eq} = \mathcal{F}^b \cup \bigcup_{c_i \in soft(\mathcal{F})} \{(\neg b_i \vee \neg \ell) : \ell \in c_i\}$$

is the relaxation of $\mathcal{F}$ with b-variable equivalences.

Again, the minimum *bcost* models of $\mathcal{F}^b_{eq}$ are MAXSAT solutions.

Proposition 2. *$mincost(\mathcal{F}) = \min_{\pi \models \mathcal{F}^b_{eq}} bcost(\pi)$. Furthermore, if $\pi \models \mathcal{F}^b_{eq}$ achieves a minimum value of $bcost(\pi)$, then π restricted to the variables of $\mathcal{F}$ is a solution for $\mathcal{F}$.*

Propositions 1 and 2 show that we can solve the MAXSAT problem $\mathcal{F}$ by searching for a *bcost* minimal satisfying assignment to $\mathcal{F}^b$ or to $\mathcal{F}^b_{eq}$. Note also that $\mathcal{F}^b_{eq}$ is a stronger theory enabling more inferences than $\mathcal{F}^b$.

3 The MAXHS Approach

In MAXHS [6] a SAT solver is called at every iteration to find a new core of $\mathcal{F}$. A MIP solver is then invoked to find a minimum *bcost* assignment to the b-variables that satisfies all of the core constraints found so far. This optimization problem corresponds to a Minimum Cost Hitting Set (MCHS) problem,[4] where the goal is to find a minimum cost set of clauses that hits each of the known cores. The optimal hitting set found by the MIP solver is tested by the SAT solver. If the SAT solver is unable to find another core, it means the MAXSAT solution has been found. Otherwise, the SAT solver returns a new core, the MIP solver re-optimizes, and the iterations continue.

This algorithm is shown in Algorithm 1. The set of cores is initialized on line 2 to the empty set. In the main loop of MAXHS, the MIP solver is invoked to find

[3] Recall we assume $hard(\mathcal{F})$ is satisfiable, so $\mathcal{F}^b$ is satisfiable.

[4] An instance of MCHS is given by a universe of weighted elements U and a collection of subsets of these elements, $\mathcal{K} = \{\kappa_1, ..., \kappa_m\}$, $\kappa_i \subseteq U$. The goal is to find a minimum weight set of elements $hs \subseteq U$ such that $hs \cap \kappa_i \neq \emptyset$ for all $\kappa_i \in \mathcal{K}$.

Algorithm 1: The MAXHS algorithm for solving MAXSAT.

```
1 MaxHS ($\mathcal{F}$)
2 $\mathcal{K} = \emptyset$
3 while true do
4     $hs$ = FindMinCostHittingSet($\mathcal{K}$)
5     (sat?,$\kappa$) = SatSolver($\mathcal{F} \setminus hs$)
      // If SAT, $\kappa$ contains the satisfying truth assignment.
      // If UNSAT, $\kappa$ is a new core.
6     if sat? then
7         break // Exit While Loop, $\kappa$ is a MAXSAT solution.
8     $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$
9 return ($\kappa$, $cost(\kappa)$)
```

a minimum cost hitting set hs of $\mathcal{K}$, on line 4. If removing hs from $\mathcal{F}$ results in a satisfiable formula (tested by the SAT solver on line 5), we break out of the loop on line 7 and return the satisfying assignment as the MAXSAT solution on line 9. Otherwise, the SAT solver will return a new core, κ to add to $\mathcal{K}$ on line 8 and the loop repeats.

The correctness of Algorithm 1 is established by the following theorem.

Theorem 1. *[6] 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 $\mathcal{F} \setminus hs$ then $mincost(\mathcal{F}) = cost(\pi) = cost(hs)$.*

This theorem shows that when Algorithm 1 breaks out of its loop, κ is a MAXSAT solution. The argument that the loop must eventually terminate is based on observing that every time the SAT solver returns a core κ, it must be distinct from all previously returned cores (because a hitting set for all previous cores does not hit κ). Since there is a finite number of distinct cores, the SAT solver must eventually be unable to find another new core and the loop will terminate.

3.1 Behaviour of MAXHS

The behaviour of MAXHS in influenced by three potential sources of exponential complexity. These include the time required by the SAT solver to solve $\mathcal{F} \setminus hs$, the time required by the MIP solver to solve the NP-hard MCHS problem, and the number of iterations required. The examples below illustrate that each of these factors can, in the worst case, cause exponential runtime.

Example 1. Let $\mathcal{F}$ be an instance of the Pigeon Hole Principle, where all clauses are considered soft with weight 1. Removing any single clause from $\mathcal{F}$ will make the remaining clauses satisfiable. Therefore, MAXHS will terminate after the first core is found. So only one MCHS problem will be solved, and it is trivial. However, the time spent by the SAT solver to find a single core will be exponential.

Example 2. Let $\mathcal{K}$ be a MCHS instance. We construct a MAXSAT instance $\mathcal{F}$ that is equivalent to $\mathcal{K}$ as follows. For each set $\kappa \in \mathcal{K}$, where $\kappa = \{e_1, ..., e_k\}$, there is a hard clause $(e_1 \vee \cdots \vee e_k)$. Finally, there is a soft clause $(\neg e)$ with weight $wt(e)$ for each element $e \in \bigcup_{\kappa \in \mathcal{K}} \kappa$. A minimal core is a core such that any proper subset is not a core. It is easy to see that the minimal cores of $\mathcal{F}$ correspond to the hard clauses of $\mathcal{F}$ and therefore the total number of minimal cores is equal to $|\mathcal{K}|$. The SAT solver can find each of the minimal cores in polynomial time, by using unit propagation alone. The number of minimal cores required by MAXHS is at most $|\mathcal{K}|$. So the only possible source of exponential runtime on $\mathcal{K}$ is solving the MCHS problems. Assuming that P $\neq$ NP, there must be some MCHS instance $\mathcal{K}$ on which MAXHS will take exponential time and this must arise when MAXHS solves the MCHS problem.

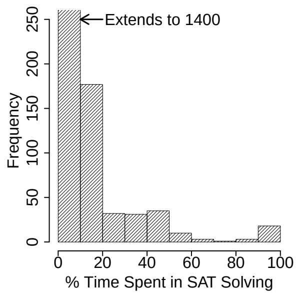
To show that exponential run time can be generated from the number of iterations required we need the following proposition.

Proposition 1 *Let n be an even number and let $E = \{e_1, ..., e_n\}$ be a universe of equally weighted elements. Let $\mathcal{K}_{n,r} = \{\kappa \subset E : |\kappa| = r\}$ be an instance of the MCHS problem where $r = \frac{n}{2}$. Let $\mathcal{K}' = \mathcal{K}_{n,r} \setminus \kappa'$ for some $\kappa' \in \mathcal{K}_{n,r}$. Then the MCHS of $\mathcal{K}'$ is strictly smaller than the MCHS of $\mathcal{K}_{n,r}$.*

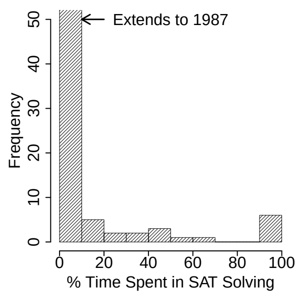
Example 3. Let $\mathcal{F}$ be a MAXSAT instance with an even number n of soft unit clauses with weight 1, $(x_1), ..., (x_n)$ and let the hard clauses of $\mathcal{F}$ form a CNF encoding of the cardinality constraint $\Sigma_{i=1}^{n} x_i < n/2$. On this family of problems, an exponential number of cores will always be required by MAXHS, as we explain next. The solutions to $\mathcal{F}$ are the truth assignments that set as many of the variables to *true* as possible without violating the hard cardinality constraint. Thus a solution to $\mathcal{F}$ will set exactly $\frac{n}{2} - 1$ of the x_i variables to *true* and the rest to *false*, and $\frac{n}{2} + 1$ is the optimal cost. Any subset of the n soft clauses, with size greater than or equal to $\frac{n}{2}$, is a core of $\mathcal{F}$. Therefore, $\mathcal{F}$ has at least $\binom{n}{n/2}$ cores. By Proposition 1, for any number of cores $k < \binom{n}{n/2}$, the cost of their MCHS is less than the optimum. Therefore, MAXHS will require at least $\binom{n}{n/2}$ cores, which is exponential in n.

However, our empirical observations are much more encouraging. In practice, we find that the SAT solving time is typically small.[5] Instead, the performance of MAXHS is most affected by the number of iterations and the time to solve the MCHS problems. Histograms of the percentage of total runtime spent by the SAT solver and the MIP solver are shown in Figure 1 over a set of 4502 Industrial and Crafted instances (the details of the experimental setup are described in Section 6). In order to study the baseline behaviour of Algorithm 1, the improvements presented in prior work [6] are omitted from this implementation. We observe in Figures 1b and 1d that on instances MAXHS failed to solve within the resource limits, the time spent by CPLEX is a much larger proportion of the

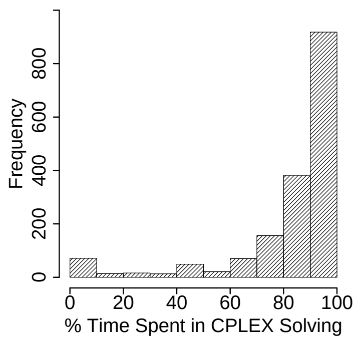
[5] If a MAXSAT instance is difficult for a state-of-the-art SAT solver to refute, then any MAXSAT solver that uses a sequence of SAT instance approach will be unable to solve it efficiently.



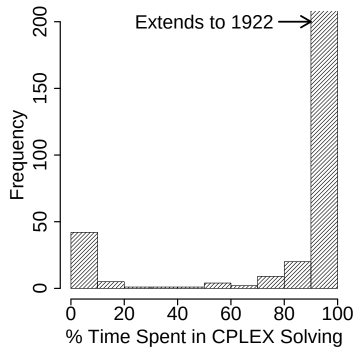
(a) Solved Instances



(b) Unsolved Instances



(c) Solved Instances



(d) Unsolved Instances

Fig. 1: Histograms over 4502 instances of the percentage of runtime spent in SAT solving and in calls to the MIP solver CPLEX, for Algorithm 1.

total runtime than the time spent on SAT solving. This is true of the solved instances as well, as shown in Figures 1a and 1c. Thus we are motivated to find ways to reduce the time spent solving the MCHS problems, since this has the greatest potential to reduce the total runtime and thus allow more instances to be solved.

4 Postponing Optimization

We have seen in the previous section that in practice, the execution time of Algorithm 1 is dominated by its multiple calls to the MIP solver. The MIP solver must optimize an NP-hard problem, Minimum Cost Hitting Set, at each iteration. Therefore, in order to improve the performance of MAXHS it is natural to ask if an approximation to MCHS can ever be used instead. In this section we show that by applying a similar approach to [16] the MAXHS algorithm can be modified to use such approximations, in order to postpone the expensive calls to the MIP solver.

Algorithm 2: An algorithm for solving MAXSAT that uses non-optimal hitting sets.

```
1  MaxHS-nonOPT ($\mathcal{F}$)
2  $\mathcal{K}$ = DisjointCores($\mathcal{F}$)
3  while true do
4      $hs$ = FindMinCostHittingSet($\mathcal{K}$)            /* Find optimal solution */
5      (sat?, $\kappa$) = SatSolver($\mathcal{F} \setminus hs$)
       // If SAT, $\kappa$ contains the satisfying truth assignment.
       // If UNSAT, $\kappa$ is a new core.
6      if sat? then
7          break  // Exit While Loop, $\kappa$ is a MAXSAT solution.
8      $\kappa$ = Minimize($\kappa$)
9      $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$
10     nonOptLevel = 0
       // Begin a series of non-optimal solutions
11     while true do
12         switch nonOptLevel do
13             case 0
14                 $hs$ = FindIncrementalHittingSet($\mathcal{K}$, $\kappa$, $hs$)
15             case 1
16                 $hs$ = FindGreedyHittingSet($\mathcal{K}$)
17         (sat?, $\kappa$) = SatSolver($\mathcal{F} \setminus hs$)
18         if sat? then
19             switch nonOptLevel do
20                 case 0
21                     nonOptLevel= 1
22                 case 1
23                     break                          /* Exit inner while loop */
24         else
25             $\kappa$ = Minimize($\kappa$)
26             $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$
27             nonOptLevel= 0
28 return ($\kappa$, $cost(\kappa)$)
```

The MAXHS algorithm from Algorithm 1 can be modified to use non-optimal hitting set computations, as shown in Algorithm 2. The algorithm operates just like MAXHS in that it terminates at line 7 if $\mathcal{F}$ is satisfiable after removing from it an **optimal** hitting set (computed at line 4). It varies from Algorithm 1 in that if a new core κ is discovered at line 5, it enters an inner loop where non-optimal hitting sets are used in place of optimal ones.

A simpler version of the algorithm uses only one method for computing approximate hitting sets rather than two. This version, which we explain first, is obtained from the version shown, by (1) replacing the **switch** statement on lines 12–16 by only one of lines 14 or 16 (i.e., we perform a single type of approximate hitting set computation); and (2) replacing the **switch** statement on lines 19–23 with a **break** statement. The resultant simpler version repeatedly finds an approximate hitting set and calls the SAT solver again to find a new core.

Eventually, the SAT solver will fail to find any more cores, the inner loop will be terminated, and we will return to line 4 where an optimal hitting set will then be computed. The simple version thus finds as many cores as possible before optimizing.

The more complex version (as shown) uses two levels of approximation: incremental and greedy. It is assumed that the second level (on line 16) computes a better (i.e., smaller) hitting set than the first. The idea here is that we compute a cheap incremental approximate hitting set until the SAT solver can't find any more cores. Then we compute the more expensive greedy approximate hitting set, which because it can be smaller might allow the SAT solver to find a new core. If a new core is found, we continue with the cheap incremental approximation (on line 27 *nonOptLevel* is reset to zero), until we once again fail to find cores with the SAT solver. If the SAT solver fails to find a new core even when using the more expensive greed approximate hitting set (line 23), the inner while loop terminates and we finally return to line 4 to compute an optimal hitting set.[6]

Two other improvements to the basic Algorithm 1 are worth mentioning (originally presented in [7]). First, at line 2, we can use the SAT solver to find a set of disjoint cores. This is accomplished by blocking every clause in the cores found so far (by setting the b-variables for the cores' clauses to *true*) and finding another core: the new core will not have any clauses in common with the previous cores. This can only be done at the start before the main loop finds other cores. Second, at lines 8 and 25, after each core is found we use the SAT solver to minimize it. This is accomplished by using a simple minimal unsatisfiable core (MUS) algorithm [17]. This results in a stronger constraint for the MIP solver.[7]

In our implementation we use two different methods for computing approximate hitting sets, both of which are very cheap. **FindIncrementalHittingSet** (line 14), simply adds a clause in the newest core to the current hitting set. The chosen clause can be any clause in κ: we choose the clause that appears most frequently in the set $\mathcal{K}$ of cores found so far. The intuition for this policy is that it takes away clauses that appear in many known cores, so that the next cores found can not use these clauses and thus are more likely to intersect with only a few of the known cores. The second method of computing non-optimal hitting sets, **FindGreedyHittingSet** (line 16), ignores the current hitting set, and instead applies a standard greedy algorithm for the MCHS problem [12].

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

Proof. This proof relies on the same argument as the correctness of Algorithm 1. If the algorithm returns on line 28, it must have broken out of the outer while loop at line 7. In this case, κ is a solution of $\mathcal{F} \setminus hs$ (by line 5), where hs is a minimum cost hitting set of $\mathcal{K}$ (by line 4). $\mathcal{K}$ is a collection of cores of $\mathcal{F}$: it is initialized on line 2, and augmented only on lines 9 and 26 with κ, a core of $\mathcal{F} \setminus hs$ (thus

[6] As can be seen from the algorithm specification we could add more cases to the **switch** statements if we had multiple approximation algorithms we wished to use.

[7] There are future possibilities for using upper and lower bounds in the algorithm (e.g., at line 17 if we find a satisfying solution its cost is an upper bound).

κ is also a core of $\mathcal{F}$). Therefore, if the algorithm returns, by Theorem 1, κ is a MAXSAT solution. It remains to show that the algorithm eventually terminates. Each time SatSolver is called (line 5 or line 17), *hs* is a hitting set of all cores in $\mathcal{K}$. So if SatSolver returns a core $\kappa \subseteq \mathcal{F} \setminus hs$ it must be distinct from all cores in $\mathcal{K}$. There is a finite number of cores, so SatSolver can not return $sat? = false$ forever and therefore both while loops must eventually terminate.

5 Additional Enhancements

In previous work we also investigated an alternative approach to reduce the time MAXHS spends in MCHS solving, that was based on using more general, **non-core**, constraints [7]. In [7] it is shown that a very effective technique is to **seed** CPLEX with many non-core constraints as a preprocessing step. In this section we show how seeding can also be applied before Algorithm 2 in order to achieve the same benefit.

Definition 3. *A non-core constraint for* MAXSAT *instance $\mathcal{F}$ is a linear inequality constraint over b-variables, c, such that $\mathcal{F}^b_{eq} \models c$.*

It is sound to add non-core constraints to the MIP model in the MAXHS algorithm. This is easy to see from Proposition 2, which states that a minimum *bcost* solution to $\mathcal{F}^b_{eq}$ corresponds to a MAXSAT solution, and the fact that the non-core constraints are entailed by $\mathcal{F}^b_{eq}$. We obtain a theorem similar to Theorem 1.

Theorem 3. *If $\mathcal{K}$ is a set of core and non-core constraints for the* MAXSAT *problem $\mathcal{F}$, π is minimum bcost assignment to the b-variables that satisfies $\mathcal{K}$, and π' is a truth assignment extending π and satisfying $\mathcal{F}^b$ then $mincost(\mathcal{F}) = bcost(\pi)$ and π' restricted to the variables of $\mathcal{F}$ is a* MAXSAT *solution.*

Proof. Since π is a minimum *bcost* assignment to the *b*-variables that satisfies $\mathcal{K}$, and all constraints in $\mathcal{K}$ are entailed by $\mathcal{F}^b_{eq}$, by Proposition 2 $mincost(\mathcal{F}) \geq bcost(\pi)$. On the other hand, π' extends π to a satisfying assignment of $\mathcal{F}^b$, and since π' sets the same *b*-variables as π, we have $bcost(\pi') = bcost(\pi)$. So by Proposition 1, $mincost(\mathcal{F}) \leq bcost(\pi') = bcost(\pi)$. Thus $mincost(\mathcal{F}) = bcost(\pi)$. Finally, π' restricted to the variables of $\mathcal{F}$ is a MAXSAT solution by Proposition 1, since π' is a minimum *bcost* satisfying assignment of $\mathcal{F}^b$.

This theorem allows us to modify Algorithm 2 by adding a preprocessing step that identifies a collection of non-core constraints $\mathcal{N}$ as shown in Algorithm 3. Now, when the MIP solver is invoked to find an optimal solution, it no longer solves a pure MCHS problem because it must take into account the seeded non-core constraints $\mathcal{N}$ in addition to the cores $\mathcal{K}$. On line 6, the MIP solver returns an optimal assignment to the *b*-variables, $\mathcal{A}$, that is then passed as a set of assumptions to the SAT solver on line 7. Note that the SAT solver uses $\mathcal{F}^b$ as input which allows the settings of the *b*-variables in $\mathcal{A}$ to relax the right set of soft clauses. By Theorem 3, if the SAT solver returns a satisfying assignment it

Algorithm 3: An algorithm for solving MAXSAT that uses non-optimal hitting sets and seeded non-core constraints.

```
1 MaxHS-nonOPT-seed ($\mathcal{F}$)
2 $\mathcal{K}$ = DisjointCores($\mathcal{F}$)
3 $\mathcal{N}$ = NonCoreConstraints($\mathcal{F}^b_{eq}$)
4 $obj = wt(c_i) * b_i + \ldots + wt(c_k) * b_k$
5 while true do
6     $\mathcal{A}$ = Optimize($\mathcal{K} \cup \mathcal{N}$, $obj$)            /* Find optimal solution */
7     (sat?, $\kappa$) = AssumptionSatSolver($\mathcal{F}^b, \mathcal{A}$)
      // Subsequent lines identical to Algorithm 2 lines 6–28
```

corresponds to the MAXSAT solution. Otherwise, the SAT solver will return a new **core** constraint κ. Thus once the main loop begins, no more non-core constraints will be derived (this differs from the previous work on non-core constraints) and the rest of the algorithm proceeds as before. The argument that the algorithm terminates remains the same as well.

Theorem 4. *Algorithm 3 returns a solution to the MAXSAT problem $\mathcal{F}$.*

It remains to specify how a collection of non-core constraints are to be found by NonCoreConstraints. We use Eq-Seeding [7] because it was found to be the most effective overall. In Eq-Seeding, we exploit the equivalence between original literals of $\mathcal{F}$ that appear in soft unit clauses, and their b-variables. In $\mathcal{F}^b_{eq}$, $b_i \equiv \neg x$ if there is a unit soft clause $(x) \in \mathcal{F}$. So to generate a collection of constraints, we consider each clause c of $\mathcal{F}^b$, and check whether each literal in c has an equivalent b-literal (or is itself a b-literal). If so, we can derive a new b-variable constraint from c by replacing every original literal by its equivalent b-literal. This constraint is a clause over the b-variables that is entailed by F^b_{eq} and it can be added to $\mathcal{N}$.

6 Experimental Results

We performed an empirical study of ten existing MAXSAT solvers: CPLEX (version 12.2), WPM1 [1], WPM2 (versions 1 and 2 [3]), BINCD [10], WBO [15], MINIMAXSAT [9], SAT4J [5], AKMAXSAT [13], MAXHS-Orig [6], and MAXHS+ [7]. All of these solvers are able to solve MAXSAT in its most general form, i.e., weighted partial MAXSAT, and thus have the widest range of applicability. Our study includes recently developed solvers utilizing a sequence of SAT approach (BINCD, WPM1, WPM2, MAXHS-Orig and MAXHS+), some older solvers (SAT4J and WBO), and two prominent Branch and Bound based solvers (AKMAXSAT and MINIMAXSAT). Also included is the MIP solver CPLEX, which is invoked after applying a standard translation of MAXSAT to MIP [7].

We experiment with three versions of Algorithm 2, that differ by how the non-optimal hitting sets are computed. The first and second versions use the

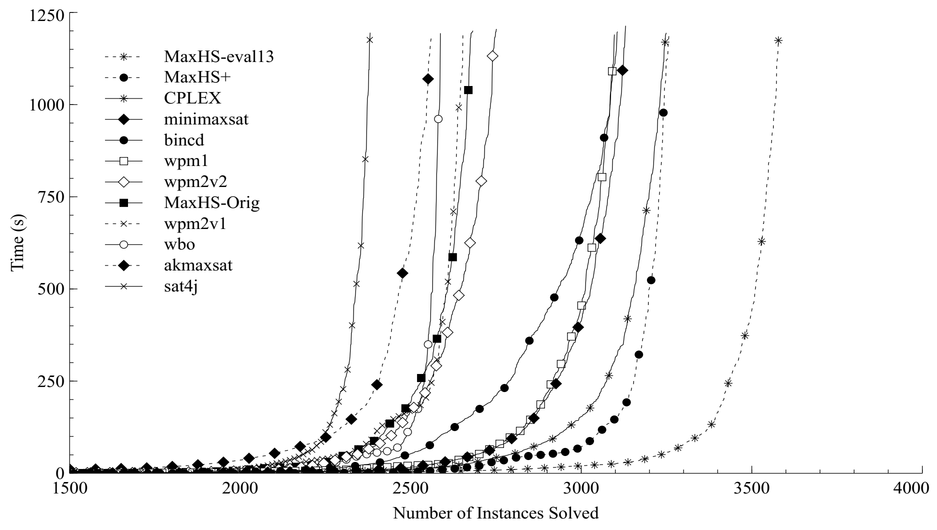


Fig. 2: Performance of solvers on all Crafted and Industrial instances.

simple form of Algorithm 2 where only one method for computing approximate hitting sets is used. The first version, **MAXHS-incr**, uses FindIncrementalHittingSet for this computation, and the second version, **MAXHS-greedy**, uses FindGreedyHittingSet. The third version is called **MAXHS-incr-greedy** and it uses the more complex version of Algorithm 2, as specified in the pseudo-code. Finally, we also experiment with the version described in Section 5 that adds Eq-Seeding to MAXHS-incr-greedy. This version uses the same algorithm as the solver submitted to the 2013 MAXSAT Evaluation, and will be called **MAXHS-eval13**.[8]

We obtained **all** problems from the previous seven MAXSAT evaluations [4], discarding all instances in the Random category. After removing duplicate problems (as many as we could find) we ended up with 4502 problems divided into 58 families.[9] The family names and the number of instances in each family are shown in Tables 1 and 2. Our experiments were performed on 2.1 GHz AMD Opteron machines with 98GB RAM shared between 24 cores (about 4GB RAM per core). Each problem was run under a 1200 second timeout and with a memory limit of 2.5GB.

The overall results are shown in Figure 2. We see that the earliest MAXHS-Orig is a reasonable but not distinguished solver. The improvement presented in [7], MAXHS+, is a very good solver being slightly better over all problems than any previous solver. Finally, we see that the best of the versions developed

[8] All versions of MAXHS in this study use MINISAT-2.0 and CPLEX version 12.2. The solver submitted to the 2013 Evaluation uses CPLEX version 12.5 and it performs slightly better than the version we report on here.

[9] We include results on 17 families within the Crafted category that were omitted in [7].

Family	#	mini	CPLEX	wpm1	bincd	MAXHS					
						Orig	+	Alg. 2			Alg. 3
								incr	grdy	i+g	eval 13
ms/Safar	112	3	19	**88**	71	75	33	33	34	33	33
ms/circdebug	9	0	1	**9**	7	**9**	3	4	3	4	3
pms/bcp-fir	59	13	**58**	53	55	16	18	32	18	33	33
pms/bcp-msp	148	108	110	60	117	62	121	90	87	95	**123**
pms/bcp-mtg	215	208	193	**215**	**215**	150	212	**215**	214	**215**	**215**
pms/bcp-syn	74	27	**71**	40	45	65	**71**	69	69	69	**71**
pms/pb/logic-syn	17	2	**16**	7	7	**16**	**16**	**16**	**16**	**16**	**16**
pms/pbo-rout	15	14	14	**15**	**15**	10	13	12	10	10	13
pms/pseudo/rout	15	14	**15**	**15**	**15**	7	**15**	11	11	10	12
pms/circtracecomp	4	1	0	3	**4**	0	0	0	1	1	1
pms/pb/primes	86	76	78	46	76	59	**80**	77	74	78	**81**
pms/pb-nencdr	128	64	23	69	116	48	104	118	**128**	**128**	127
pms/pb-nlogencdr	128	103	24	88	**128**	78	111	**128**	**128**	**128**	**128**
pms/aes	7	1	2	0	1	1	2	2	2	**3**	2
pms/hap-asmbly	6	0	2	4	0	**5**	**5**	**5**	**5**	**5**	**5**
pms/bcp-hipp	1183	982	962	1154	**1164**	1125	1142	1141	1138	1137	1140
wpms/haplo-ped	100	0	9	**91**	23	27	28	26	33	25	22
pms/protein-ins	12	**11**	1	1	2	1	1	2	1	2	2
wpms/protein-ins	12	**10**	1	1	2	1	2	2	1	2	1
wpms/timetabling	32	0	0	**13**	12	9	8	7	7	7	6
wpms/upgrade	100	0	**100**	**100**	97	**100**	**100**	**100**	**100**	**100**	**100**
wpms/up-u98	80	0	**80**	**80**	79	**80**	**80**	**80**	**80**	**80**	**80**
Total	2542	1637	1779	2152	**2251**	1944	2165	2170	2160	2181	2214
Indust + Craft Total	4502	3130	3249	3097	3106	2682	3257	3288	3297	3419	**3578**

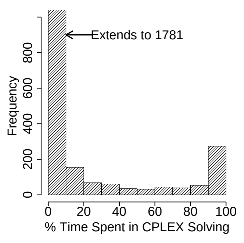
Table 1: Results for the Industrial category instances. Shows the number of instances solved by each solver in each benchmark family. The final row gives the total number of instances solved over both the Industrial and Crafted categories.

in this paper, MAXHS-eval13 achieves a significant performance improvement. Although not shown on the plot, the other versions we develop here all improve over MAXHS+, but are not as good as MAXHS-eval13 (see Tables 1 and 2).

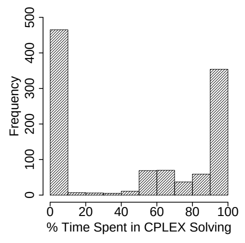
The results are broken down by benchmark family in Tables 1 and 2. Included in the tables are the four competing solvers that performed best overall (as shown in Figure 2): CPLEX, MINIMAXSAT, BINCD and WPM1. Observe that all versions of MAXHS that use non-optimal hitting sets (Alg. 2 and Alg. 3) outperform MAXHS+, which uses non-core constraints more extensively than MAXHS-eval13. Yet when the technique of seeding the MIP model with non-core constraints is added to the use of non-optimal hitting sets, performance is improved (see "Alg. 2 i+g" vs. "Alg. 3 eval 13"). We see that on the industrial problems (Table 1) there is still quite a lot of variance in performance between the different solvers across the different families; that MAXHS-eval13 has fairly robust good performance across the different families; and that the MAXHS approach can be significantly better than just using CPLEX alone, indicating the value of our hybrid approach. On the crafted problems (Table 2) we see that the Branch and Bound approach of MINIMAXSAT is most effective. These problems tend to be smaller than the industrial problems and have tightly interacting variables yielding cores containing a large fraction of the total clauses. The data also

Family	#	bincd	wpm1	CPLEX	mini	MAXHS					
						Orig	+	Alg. 2			Alg. 3
								incr	grdy	i+g	eval 13
*ms/spin	20	0	0	19	**20**	0	0	1	4	10	10
*ms/cut/spin	5	1	1	**3**	**3**	0	1	1	2	2	2
*wms/cut/spin	5	0	0	**4**	**4**	0	1	1	2	3	3
*pms/frb	25	0	0	**9**	5	0	8	5	0	5	**9**
*wms/kexu/frb	35	9	5	**20**	15	10	**20**	16	15	15	**20**
*pms/csp/sprsls	20	**20**	**20**	**20**	**20**	19	11	**20**	**20**	**20**	**20**
*pms/csp/dsls	20	**20**	16	**20**	**20**	5	0	15	15	16	16
*pms/csp/sprstgt	20	**20**	**20**	**20**	**20**	0	0	15	12	**20**	**20**
*pms/csp/dstgt	20	19	19	**20**	**20**	0	0	3	6	**20**	**20**
*ms/ramsey	48	34	34	34	**35**	34	34	34	34	34	34
*wms/ramsey	48	36	34	36	**37**	34	35	34	34	34	34
*pms/clq/rand	96	67	0	**96**	**96**	4	**96**	4	44	59	**96**
*ms/bcut-630	100	0	0	0	**83**	0	0	0	0	0	0
*pms/kbtree	54	15	14	**54**	22	11	15	12	12	12	14
*pms/max1/3sat	80	**80**	71	**80**	**80**	20	**80**	45	44	57	**80**
*ms/cut/rand	40	0	0	4	**40**	0	0	0	0	0	0
*wms/cut/rand	40	0	0	12	**40**	0	0	0	0	0	0
ms/cut/dimacs	62	6	5	20	**48**	4	4	4	4	4	3
wms/cut/dimacs	62	4	5	22	**55**	3	3	4	5	8	9
pms/max1/struc	60	59	30	52	**60**	5	**60**	54	57	**60**	**60**
pms/clique/struc	62	18	8	32	**36**	10	29	12	17	17	34
pms/queens	7	**7**	**7**	**7**	**7**	2	3	5	4	5	5
wpms/QCP	25	**25**	**25**	**25**	20	**25**	**25**	**25**	**25**	**25**	23
pms/pb/gardn	7	5	5	**6**	5	5	**6**	5	**6**	**6**	**6**
wpms/pb/mip	16	**7**	6	6	5	6	**7**	**7**	**7**	**7**	5
wpms/pb/factor	186	**186**	168	**186**	**186**	**186**	**186**	**186**	**186**	**186**	172
wpms/KnotPip	350	0	161	245	117	57	52	**290**	260	**290**	289
wpms/spot5log	21	11	**12**	6	4	6	6	6	6	6	6
wpms/spot5dir	21	11	10	**17**	3	6	6	6	6	6	6
pms/jobshop	4	**4**	3	0	2	**4**	3	**4**	**4**	**4**	**4**
wpms/plan	71	65	64	70	**71**	46	**71**	**71**	**71**	**71**	61
wpms/aucreg	84	6	0	**84**	**84**	34	**84**	4	13	2	76
wpms/aucsch	84	66	**84**	**84**	**84**	82	**84**	76	77	78	75
wpms/aucpath	88	0	52	**88**	**88**	**88**	**88**	**88**	**88**	**88**	78
wpms/ware	18	1	14	**18**	2	1	**18**	9	1	12	**18**
wpms/plan	56	53	52	51	**56**	31	**56**	**56**	**56**	**56**	**56**
Total	1960	855	945	1470	**1493**	738	1092	1118	1137	1238	1364
Indust + Craft Total	4502	3106	3097	3249	3130	2682	3257	3288	3297	3419	**3578**

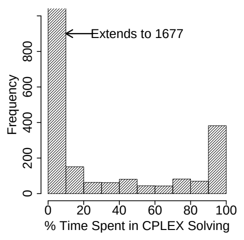
Table 2: Results for the Crafted category instances. Shows the number of instances solved by each solver in each benchmark family. The benchmark families with an asterisk (*) are those we classify as having "random" structure [7]. The final row gives the total number of instances solved over both the Industrial and Crafted categories.



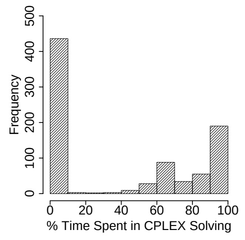
(a)



(b)



(c)



(d)

Fig. 3: Histograms over all instances for the percentage of runtime spent in calls to CPLEX for Algorithms 2 and 3. (a) Alg. 2, solved instances; (b) Alg. 2, unsolved instances; (c) Alg. 3, solved instances; (d) Alg. 3, unsolved instances.

shows that although the traditional sequence of SAT solvers BINCD and WPM1 do not perform particularly well on these problems, CPLEX is quite effective, as is the hybrid approach of MAXHS.

The good performance of MAXHS-incr-greedy and MAXHS-eval13, implementing Algorithms 2 and 3 respectively, appears to be due to a significant reduction in the total time spent by CPLEX. In Figure 3 we show the percentage of the total runtime that was spent in calls to CPLEX. Comparing these histograms to those in Figure 1, we observe that the time spent solving the MCHS problems to optimality is now almost always a low percentage of the total runtime. The number of calls to CPLEX generally decreases when we use non-optimal hitting sets, as expected. On average (over all 4502 instances), each run of MAXHS-incr-greedy gave a total of 5419 constraints to CPLEX but solved the optimization problem only 14 times. In contrast, each run of Algorithm 1 gave on average only 972 cores to CPLEX, thus having to solve the MCHS problem 972 times.

In conclusion, we have presented a technique for improving the hybrid MAXHS approach to solving MAXSAT. Our method yields an improvement in the state-of-the-art for MAXSAT solving. Although our method successfully shifts the balance of the runtime away from the MIP solver, a promising avenue for future work is to examine the structure of the constraints given to the MIP solver to see if they could be made more effective.

Acknowledgements This work has been partly funded by the "Agence nationale de la Recherche", reference ANR-10-BLA-0214.

References

1. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving sat-based weighted maxsat solvers. In: Principles and Practice of Constraint Programming (CP). pp. 86–101 (2012)
2. 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)
3. 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)
4. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The maxsat evaluations (2007–2011), `http://www.maxsat.udl.cat`
5. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT 7(2-3), 59–6 (2010)
6. Davies, J., Bacchus, F.: Solving maxsat by solving a sequence of simpler sat instances. In: Principles and Practice of Constraint Programming (CP). pp. 225–239 (2011)
7. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT) (2013)
8. Eén, N., Sörensson, N.: An extensible sat-solver. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT). pp. 502–518 (2003)
9. Heras, F., Larrosa, J., Oliveras, A.: Minimaxsat: An efficient weighted max-sat solver. Journal of Artificial Intelligence Research (JAIR) 31, 1–32 (2008)
10. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Proceedings of the AAAI National Conference (AAAI). pp. 36–41 (2011)
11. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. Operations Research 55(3), 588–602 (2007)
12. Johnson, D.S.: Approximation algorithms for combinatorial problems. In: Symposium on Theory of Computing. pp. 38–49 (1973)
13. Kügel, A.: Improved exact solver for the weighted Max-SAT problem. In: Workshop on the Pragmatics of SAT (2010)
14. Li, C.M., Manyà, F., Mohamedou, N.O., Planes, J.: Resolution-based lower bounds in maxsat. Constraints 15(4), 456–484 (2010)
15. 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)
16. Moreno-Centeno, E., Karp, R.M.: The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. Operations Research 61(2), 453–468 (March-April 2013)
17. Silva, J.P.M., Lynce, I.: On improving mus extraction algorithms. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT). pp. 159–173 (2011)