Solving MaxSAT by Successive Calls to a SAT Solver

Mohamed El Halaby(✉)

Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt
halaby@sci.cu.edu.eg

Abstract. The Maximum Satisfiability (MaxSAT) is the problem of determining the maximum number of clauses of a given Boolean formula in Conjunctive Normal Form (CNF) that can be satisfied by an assignment of truth values to the variables of the formula. Many exact solvers for MaxSAT have been developed during recent years, and many of them were presented in the well-known SAT Conference. Algorithms for MaxSAT generally fall into two categories: (1) branch and bound algorithms and (2) algorithms that use successive calls to a SAT solver (SAT-based), which this paper in on. In practical problems, SAT-based algorithms have been shown to be more efficient. This paper provides an experimental investigation to compare the performance of recent SAT-based and branch and bound algorithms on benchmarks of the MaxSAT Evaluations.

Keywords: MaxSAT · Satisfiability · Boolean logic

1 Introduction

A *Boolean variable* x can take one of two possible values 0 (false) or 1 (true). A *literal* l is a variable x or its negation $\neg x$. A *clause* is a disjunction of literals, i.e., $\bigvee_{i=1}^{n} l_i$. A *CNF formula* is a conjunction of clauses. Formally, a CNF formula ϕ composed of k clauses, where each clause C_i is composed of m_i is defined as $F = \bigwedge_{i=1}^{k} C_i$ where $C_i = \bigvee_{j=1}^{m_i} l_{i,j}$.

In this paper, a set of clauses $\{C_1, C_2, \ldots, C_k\}$ is referred to as a Boolean formula. A truth assignment *satisfies* a Boolean formula if it satisfies every clause.

Given a CNF formula ϕ, the *Maximum Satisfiability* (MaxSAT) problem asks for a truth assignment that maximizes the number of satisfied clauses in ϕ.

Let $\phi = \{(C_1, w_2), \ldots, (C_s, w_s)\} \cup \{(C_{s+1}, \infty), \ldots, (C_{s+h}, \infty)\}$ be a CNF formula, where $w_1, \ldots, w_s$ are natural numbers. The Weighted Partial MaxSAT problem asks for an assignment that satisfies all $C_{s+1}, \ldots, C_{s+h}$ (called *hard* clauses) and maximizes the sum of the weights of the satisfied clauses in $C_1, \ldots, C_s$ (called *soft* clauses).

The SAT-based approach to solving MaxSAT converts the instance into a sequence of SAT instances which can be solved using SAT solvers. One way to do this, given an MaxSAT instance, is to check if there is an assignment that

Y. Bi et al. (eds.), *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016*, Lecture Notes in Networks and Systems 15, DOI 10.1007/978-3-319-56994-9_31

falsifies no clauses. If such an assignment can not be found, we check if there is an assignment that falsifies only one clause. This is repeated and each time we increment the number of clauses that are allowed to be $False$ until the SAT solver returns $True$, meaning that the minimum number of falsified clauses has been determined. Recent comprehensive surveys on SAT-based algorithms can be found in [5,18].

The paper consists of two main parts: (1) Detailed descriptions of MaxSAT algorithms based on calling a SAT solver (SAT-based algorithms). (2) Experimental investigation and comparison between SAT-based MaxSAT solvers and branch-and-bound solvers. In this part, solvers from both categories were run on a wide set of benchmarks from the MaxSAT Evaluations and in each benchmark category, a winner solver is declared. Such an investigation is crucial in order to test the limits of MaxSAT solving techniques and to better adapt MaxSAT technologies to practical frontiers.

2 Linear Search Algorithms

A simple way to solve MaxSAT is to augment each soft clause C_i with a new variable (called a blocking variable) b_i, then a constraint is added (specified in CNF) saying that the sum of the weights of the falsified soft clauses must be less than a given value k. Next, the formula (without the weights) together with the constraint is sent to a SAT solver to check whether or not it is satisfiable. If so, then the cost of the optimal solution is found and the algorithm terminates. Otherwise, k is decreased and the process continues until the SAT solver returns $True$. The algorithm can start searching for the optimal cost from a lower bound LB initialized with the maximum possible cost (i.e. $LB = \sum_{i=1}^{|\phi_S|} w_i$) and decrease it down to the optimal cost, or it can set $LB = 0$ and increase it up to the optimal cost. Solvers that employ the former approach is called *satisfiability-based*

Algorithm 1. LinearUNSAT(ϕ) Linear search UNSAT-based algorithm for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A MaxSAT solution to ϕ

```
1 LB ← 0
2 foreach (C_i, w_i) ∈ φ_S do
3     let b_i be a new blocking variable
4     φ_S ← φ_S \ {(C_i, w_i)} ∪ {(C_i ∨ b_i, w_i)}
5 while True do
6     (state, I) ← SAT({C | (C, w) ∈ φ} ∪ CNF(Σ_{i=1}^{|φ_S|} w_i b_i ≤ LB))
7     if state = True then
8         return I
9     LB ← UpdateBound({w | (C, w) ∈ φ_S}, LB)
```

Algorithm 2. LinearSAT(ϕ) Linear search SAT-based algorithm for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A MaxSAT solution to ϕ
1 $UB \leftarrow 1 + \sum_{i=1}^{|\phi_S|} w_i$
2 **foreach** $(C_i, w_i) \in \phi_S$ **do**
3 let b_i be a new blocking variable $\phi_S \leftarrow \phi_S \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$
4 **while** $True$ **do**
5 $(state, I) \leftarrow SAT(\{C \mid (C, w) \in \phi\} \cup CNF(\sum_{i=1}^{|\phi_S|} w_i b_i \leq UB - 1))$
6 **if** $state = False$ **then**
7 **return** $lastI$
8 $lastI \leftarrow I$
9 $UB \leftarrow \sum_{i=1}^{|\phi_S|} w_i(1 - I(C_i \setminus \{b_i\}))$

(not to be confused with the name of the general method) solvers, while the ones that follow the latter are called *UNSAT-based* solvers. A cost of 0 means all the soft clauses are satisfied and a cost of means all the soft clauses are falsified. Algorithms 1 and 2 employ these techniques ti find the optimal cost.

Note that updating the upper bound to $\sum_{i=1}^{|\phi_S|} w_i(1-I(C_i \setminus \{b_i\}))$ is more efficient than simply decreasing the upper bound by one, because uses less iterations and thus the problem is solved with less SAT calls.

3 Binary Search Based Algorithms

In the worst case the linear search can take $\sum_{i=1}^{|\phi_S|} w_i$ calls to the SAT solver. Since we are searching for a value (the optimal cost) among a set of values (from 0 to $\sum_{i=1}^{|\phi_S|} w_i$), then binary search can be used, which uses less iterations than linear search. Algorithm 3 searches for the cost of the optimal assignment by using binary search.

BinS-MaxSAT begins by checking the satisfiability of the hard clauses (line 1) before beginning the search for the solution. If the SAT solver returns $False$ (line 2), BinS-MaxSAT returns the empty assignment and terminates (line 3). The algorithm updates both a lower bound LB and an upper bound UB initialized respectively to -1 and one plus the sum of the weights of the soft clauses (lines 4–5). The soft clauses are augmented with blocking variables (lines 6–8). At each iteration of the main loop (lines 9–16), the middle value (mid) is changed to the average of LB and UB and a constraint is added requiring the sum of the weights of the relaxed soft clauses to be less than or equal to the middle value. This clauses describing this constraint are sent to the SAT solver along with the clauses of ϕ (line 11). If the SAT solver returns $True$ (line 12), then the cost of the optimal solution is less than mid, and UB is updated (line 14). Otherwise, the algorithm looks for the optimal cost above mid, and so LB is

Algorithm 3. BinS-MaxSAT(ϕ) Binary search based algorithm for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A MaxSAT solution to ϕ

```
1  state ← SAT({C_i | (C_i, ∞) ∈ φ_H})
2  if state = False then
3      return ∅
4  LB ← −1
5  UB ← 1 + Σ_{i=1}^{|φ_S|} w_i
6  foreach (C_i, w_i) ∈ φ_S do
7      let b_i be a new blocking variable
8      φ_S ← φ_S \ {(C_i, w_i)} ∪ {(C_i ∨ b_i, w_i)}
9  while LB + 1 < UB do
10     mid ← ⌊(LB+UB)/2⌋
11     (state, I) ← SAT({C | (C, w) ∈
       φ} ∪ CNF(Σ_{i=1}^{|φ_S|} w_i b_i ≤ mid))
12     if state = True then
13         lastI ← I
14         UB ← Σ_{i=1}^{|φ_S|} w_i(1 − I(C_i \ {b_i}))
15     else
16         LB ← UpdateBound({w_i | 1 ≤ i ≤ |φ_S|}, mid) − 1
17 return lastI
```

updated (line 16). The main loop continues until $LB+1 = UB$, and the number of iterations BinS-MaxSAT executes is proportional to $\log(\sum_{i=1}^{|\phi_S|} w_i)$ which is a considerably lower complexity than that of linear search methods.

Algorithm 4 begins by checking that the set of hard clauses is satisfiable (line 1). If not, then the algorithm returns the empty assignment and terminates (line 3). Next, the soft clauses are relaxed (lines 4–6) and the lower and upper bounds are initialized respectively to -1 and one plus the sum of the weights of the soft clauses (lines 7–8). BinLin-MaxSAT has two execution modes, binary and linear. The mode of execution is initialized in line 9 to binary search. At each iteration of the main loop (lines 10–27), the SAT solver is called on the clauses of ϕ with the constraint $\sum_{i=1}^{|\phi_S|} w_i b_i$ bounded by the mid point (line 12), if the current mode is binary, or by the upper bound if the mode is linear (line 14). If the formula is satisfiable (line 16), the upper bound is updated. Otherwise, the lower bound is updated to the mid point. At the end of each iteration, the mode of execution is flipped (lines 24–27).

Since the cost of the optimal solution is an integer, it can be represented as an array of bits. Algorithm 5 uses this fact to determine the solution bit by bit. BitBased-MaxSAT starts from the most significant bit and at each iteration it moves one bit closer to the least significant bit, at which the optimal cost if found.

Algorithm 4. BinLin-MaxSAT(ϕ) Alternating binary and linear searches for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A MaxSAT solution to ϕ

1 $state \leftarrow SAT(\{C_i \mid (C_i, \infty) \in \phi_H\})$
2 **if** $state = False$ **then**
3 **return** $\emptyset$
4 **foreach** $(C_i, w_i) \in \phi_S$ **do**
5 let b_i be a new blocking variable
6 $\phi_S \leftarrow \phi_S \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$
7 $LB \leftarrow -1$
8 $UB \leftarrow 1 + \sum_{i=1}^{|\phi_S|} w_i$
9 $mode \leftarrow binary$
10 **while** $LB + 1 < UB$ **do**
11 **if** $mode = binary$ **then**
12 $mid \leftarrow \lfloor \frac{LB+UB}{2} \rfloor$
13 **else**
14 $mid \leftarrow UB - 1$
15 $(state, I) \leftarrow SAT(\{C \mid (C, w) \in \phi\} \cup CNF(\sum_{i=1}^{|\phi_S|} w_i b_i \leq mid))$
16 **if** $state = True$ **then**
17 $lastI \leftarrow I$
18 $UB \leftarrow \sum_{i=1}^{|\phi_S|} w_i(1 - I(C_i \setminus \{b_i\}))$
19 **else**
20 **if** $mode = binary$ **then**
21 $LB \leftarrow UpdateBound(\{w_i \mid 1 \leq i \leq |\phi_S|\}, mid) - 1$
22 **else**
23 $LB \leftarrow mid$
24 **if** $mode = binary$ **then**
25 $mode \leftarrow linear$
26 **else**
27 $mode \leftarrow binary$
28 **return** $lastI$

The satisfiability of the hard clauses is checked and the soft clauses are relaxed. The sum of the weights of the soft clauses k is an upper bound on the cost and thus it is computed to determine the number of bits needed to represent the optimal solution (line 7). The index of the current bit being considered is initialized to k (line 7), and the value of the solution being constructed is initialized (line 8). The main loop (lines 10–20) terminates when it reached the least significant bit (when $CurrBit = 0$). At each iteration, the SAT solver is called on ϕ with constraint saying that the sum of the weights of the falsified soft

Algorithm 5. BitBased-MaxSAT(ϕ) A bit-based algorithm for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A MaxSAT solution to ϕ

1 $state \leftarrow SAT(\{C_i \mid (C_i, \infty) \in \phi_H\})$
2 **if** $state = False$ **then**
3 **return** $\emptyset$
4 **foreach** $(C_i, w_i) \in \phi_S$ **do**
5 let b_i be a new blocking variable
6 $\phi_S \leftarrow \phi_S \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$
7 $k \leftarrow \lfloor \lg(\sum_{i=1}^{|\phi_S|} w_i) \rfloor$
8 $CurrBit \leftarrow k$
9 $cost \leftarrow 2^k$
10 **while** $CurrBit \geq 0$ **do**
11 $(state, I) \leftarrow SAT(\{C \mid (C, w) \in \phi\} \cup CNF(\sum_{i=1}^{|\phi_S|} w_i b_i < cost))$
12 **if** $state = True$ **then**
13 $lastI \leftarrow I$
14 let $s_0, \ldots, s_k \in \{0, 1\}$ be constants such that $\sum_{i=1}^{|\phi_S|} w_i(1 - I(C_i \setminus \{b_i\})) = \sum_{j=0}^{k} 2^j s_j$
 $CurrBit \leftarrow max(\{j \mid j < CurrBit \text{ and } s_j = 1\} \cup \{-1\})$
15 **if** $CurrBit \geq 0$ **then**
16 $cost \leftarrow \sum_{j=CurrBit}^{k} 2^j s_j$
17 **else**
18 $CurrBit \leftarrow CurrBit - 1$
19 $cost \leftarrow cost + 2^{CurrBit}$
20 **return** $lastI$

clauses must be less than *cost* (line 11). If the SAT solver returns $True$ (line 12), the sum of the weights of the soft clauses falsified by the current assignment is computed and the set of bits needed to represent that number are determined as well (line 14), the index of the current bit is decreased to the next $j < CurrBit$ such that $s_j = 1$ (line 15). If such an index does not exist, then $CurrBit$ becomes -1 and in the following iteration the algorithm terminates. On the other hand, if the SAT solver returns $False$, the search continues to the most significant bit by decrementing $CurrBit$ (line 19) and since the optimal cost is greater than the current value of *cost*, it is decreased by $2^{CurrBit}$ (line 20).

4 Core-Guided Algorithms

As in the previous method, UNSAT methods use SAT solvers iteratively to solve MaxSAT. Here, the purpose of iterative SAT calls is to identify and relax unsatisfiable formulas (unsatisfiable cores) in a MaxSAT instance. This method was

first proposed in 2006 by Fu and Malik in [9] (see Algorithm 6). The algorithms described in this section are

(1) Fu and Malik's algorithm [9]
(2) WPM1 [1]
(3) Improved WPM1 [2]
(4) WPM2 [4]
(5) WMSU4 [17]

Definition 4.1 (Unsatisfiable core)**.** *An unsatisfiable core of a CNF formula ϕ is a subset of ϕ that is unsatisfiable by itself.*

Definition 4.2 (Minimum unsatisfiable core)**.** *A minimum unsatisfiable core contains the smallest number of the original clauses required to still be unsatisfiable.*

Definition 4.3 (Minimal unsatisfiable core)**.** *A minimal unsatisfiable core is an unsatisfiable core such that any proper subset of it is not a core [7].*

Modern SAT solvers provide the unsatisfiable core as a by-product of the proof of unsatisfiability.

The idea in this paradigm is as follows: Given a MaxSAT instance $\phi = \{(C_1, w_1), \ldots, (C_s, w_s)\} \cup \{(C_{s+1}, \infty), \ldots, (C_{s+h}, \infty)\}$, let ϕ_k be a SAT instance that is satisfiable iff ϕ has an assignment with cost less than or equal to k. To encode ϕ_k, we can extend every soft clause C_i with a new (auxiliary) variable b_i and add the CNF conversion of the constraint $\sum_{i=1}^{s} w_i b_i \leq k$. So, we have $\phi_k = \{(C_i \vee b_i), \ldots, (C_s \vee b_s), C_{s+1}, \ldots, C_{s+h}\} \cup CNF\,(\sum_{i=1}^{s} w_i b_i \leq k)$.

Let k_{opt} be the cost of the optimal assignment of ϕ. Thus, ϕ_k is satisfiable for all $k \geq k_{opt}$, and unsatisfiable for all $k < k_{opt}$, where k may range from 0 to $\sum_{i=1}^{s} w_i$. Hence, the search for the optimal assignment corresponds to the location of the transition between satisfiable and unsatisfiable ϕ_k. This encoding guarantees that the all the satisfying assignments (if any) to $\phi_{k_{opt}}$ are the set of optimal assignments to the MaxSAT instance ϕ.

4.1 Fu and Malik's Algorithm

Fu and Malik implemented two PMaxSAT solvers, ChaffBS (uses binary search to find the optimal cost) and ChaffLS (uses linear search to find the optimal cost) on top of a SAT solver called zChaff [19]. Their PMaxSAT solvers participated in the first and second MaxSAT Evaluations. Their method (Algorithm 6 basis for many MaxSAT solvers that came later. Notice the input to Algorithm 6 is a PMaxSAT instance since all the weights of the soft clauses are the same.

Fu and Malik (Algorithm 6) (also referred to as MSU1) begins by checking if a hard clause is falsified (line 1), and if so it terminates returning the cost ∞ (line 2). Next, unsatisfiable cores (ϕ_C) are identified by iteratively calling a SAT solver on the soft clauses (line 6). If the working formula is satisfiable (line 7), the algorithm halts returning the cost of the optimal assignment (line 8). If not,

Algorithm 6. Fu and Malik(ϕ) Fu and Malik's algorithm for solving PMaxSAT.

Input: $\phi = \{(C_1, 1), \ldots, (C_s, 1), (C_{s+1}, \infty), \ldots, (C_{s+h}, \infty)\}$
Output: The cost of the optimal assignment to ϕ

```
1  if SAT({C_{s+1}, ..., C_{s+h}}) = (False, _) then
2      return ∞
3  opt ← 0 f ← 0 while True do
4      (state, φ_C) ← SAT({C_i | (C_i, w_i) ∈ φ})
5      if state = True then
6          return opt
7      f ← f + 1
8      B ← ∅
9      foreach C_i ∈ φ_C such that w_i ≠ ∞ do
10         let b_i be a new blocking variable
11         φ ← φ \ {(C_i, 1)} ∪ {(C_i ∨ b_i, 1)}
12         B ← B ∪ {i}
13     φ ← φ ∪ {(C, ∞) | C ∈ Σ_{i∈B} b_i = 1} opt ← opt + 1
```

then the algorithm starts its second phase by relaxing each soft clause in the unsatisfiable core obtained earlier by adding to it a fresh variable, in addition to saving the index of the relaxed clause in B (lines 11–14). Next, the new working formula constraints are added indicating that exactly one of b_i variables should be $True$ (line 15). Finally, the cost is increased by one (line 16) a clause is falsified. This procedure continues until the SAT solver declares the formula satisfiable.

4.2 WPM1

Ansótegui, Bonet and Levy [1] extended Fu and Malik to MaxSAT. The resulting algorithm is called WPM1 and is described in Algorithm 7.

Just as in Fu and Malik, Algorithm 7 calls a SAT solver iteratively with the working formula, but without the weights (line 5). After the SAT solver returns an unsatisfiable core, the algorithm terminates if the core contains hard clauses and if it does not, then the algorithm computes the minimum weight of the clauses in the core, w_{min} (line 9). Next, the working formula is transformed by duplicating the core (line 13) with one copy having the clauses associated with the original weight minus the minimum weight and a second copy having having the clauses augmented with blocking variables with the original weight. Finally, the cardinality constraint on the blocking variable is added as hard clauses (line 18) and the cost is increased by the minimum weight (line 19).

WPM1 uses blocking variables in an efficient way. That is, if an unsatisfiable core, $\phi_C = \{C_1, \ldots, C_k\}$, appears l times, all the copies get the same set of blocking variables. This is possible because the two formulae $\phi_1 = \phi \setminus \phi_C \cup \{C_1 \vee b_i, \ldots, C_i \vee b_i \mid C_i \in \phi_C\} \cup CNF\left(\sum_{i=1}^{k} b_i = 1\right)$ and $\phi_2 = \phi \setminus \phi_C \cup \{C_i \vee b_i^1, \ldots, C_i \vee b_i^l \mid$

Algorithm 7. WPM1(ϕ) The WPM1 algorithm for MaxSAT.

Input: A MaxSAT instance
$\phi = \{(H_1, \infty), \ldots, (H_h, \infty)\} \cup \{(S_1, w_1), \ldots, (S_s, w_s)\}$
Output: The optimal cost of the MaxSAT solution

```
1  if SAT({H_i | 1 ≤ i ≤ h}) = False then
2      return ∞
3  cost ← 0
4  while True do
5      (state, φ_C) ← SAT({C_i | (C_i, w_i) ∈ φ})
6      if state = True then
7          return cost
8      BV ← ∅
9      w_min ← min{w_i | C_i ∈ φ_C and w_i ≠ ∞}
10     foreach C_i ∈ φ_C do
11         if w_i ≠ ∞ then
12             Let b_i be a new blocking variable
13             φ ← φ \ {(C_i, w_i)} ∪ {(C_i, w_i − w_min)} ∪ {(C_i ∨ b_i, w_min)}
14             BV ← BV ∪ {b_i}
15     if BV = ∅ then
16         return False
17     else
18         φ ← φ ∪ CNF(∑_{b∈BV} b = 1)
19     cost ← cost + w_min
```

$C_i \in \phi_C\} \cup CNF\left(\sum_{i=1}^{k} b_i^1 = 1\right) \cup \cdots \cup CNF\left(\sum_{i=1}^{k} b_i^l = 1\right)$ are MaxSAT equivalent, meaning that the minimum number of unsatisfiable clause of ϕ_1 and ϕ_2 is the same. However, the algorithm does not avoid using more than one blocking variable per clause.

Example 4.1. *Consider* $\phi = \{(x_1, 1), (x_2, 2), (x_3, 3), (\neg x_1 \vee \neg x_2, \infty), (x_1 \vee \neg x_3, \infty), (x_2 \vee \neg x_3, \infty)\}$. *In the following,* b_i^j *is the relaxation variable added to clause* C_i *at the* j*th iteration. A possible execution sequence of the algorithm is:*

(1) $state = False$, $\phi_C = \{(\neg x_3), (\neg x_1 \vee \neg x_2), (x_1 \vee \neg x_3), (x_2 \vee \neg x_3)\}$, $w_{min} = 3$, $\phi = \{(x_1, 1), (x_2, 2), (x_3 \vee b_3^1, 3),, (\neg x_1 \vee \neg x_2, \infty), (x_1 \vee \neg x_3, \infty), (x_2 \vee \neg x_3, \infty), (b_3^1 = 1, \infty)\}$.
(2) $state = False$, $\phi_C = \{(x_1), (x_2), (\neg x_1 \vee \neg x_2)\}$, $w_{min} = 1$, $\phi = \{(x_1 \vee b_1^2), (x_2, 1), (x_2 \vee b_2^2), (x_3 \vee b_3^1), (\neg x_1 \vee \neg x_2, \infty), (x_1 \vee \neg x_3, \infty), (x_2 \vee \neg x_3, \infty), (b_3^1 = 1, \infty), (b_1^2 + b_2^2 = 1, \infty)$.
(3) $state = True$, $A = \{x_1 = 0, x_2 = 1, x_3 = 0\}$ *is an optimal assignment.*

If the SAT solver returns a different unsatisfiable core in the first iteration, a different execution sequence is going to take place.

4.3 Improved WPM1

In 2012, Ansótegui, Bonet and Levy presented a modification to WPM1 (Algorithm 7) [2]. In WPM1, the clauses of the core are duplicated after computing their minimum weight w_{min}. Each clause C_i in the core, the $(C_i, w_i - w_{min})$ and $(C_i \vee b_i, w_{min})$ are added to the working formula and (C_i, w_i) is removed. This process of duplication can be inefficient because a clause with weight w can be converted into w copies with weight 1. The authors provided the following example to illustrate this issue: consider $\phi = \{(x_1, 1), (x_2, w), (\neg x_2, \infty)\}$. If the SAT solver always includes the first clause in the identified core, the working formula after the first iteration will be $\{(x_1 \vee b_1^1, 1), (x_2 \vee b_2^1, 1), (x_2, w-1), (\neg x_2, \infty), (b_1^1 + b_2^1 = 1, \infty)\}$. If at each iteration i, the SAT solver includes the first clause and with $\{(x_2, w-i+1), (\neg x_2, \infty)\}$ in the unsatisfiable core, then after i iterations the formula would be $\{(x_1 \vee b_1^1 \vee \cdots \vee b_1^i, 1), (x_2 \vee b_2 * 1, 1), \ldots, (x_2 \vee b_2^i, 1), (x_2, w-i), (\neg x_2, \infty), (b_1^1 + b_2^1 = 1, \infty), \ldots, (b_1^i + b_2^i = 1, \infty)\}$. In this case, WPM1 would need w iterations to solve the problem.

Algorithm 8. ImprovedWPM1(ϕ) The stratified approach for WPM1 algorithm.

Input: A MaxSAT instance
$\phi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, w_{m+m'})\}$
Output: The cost of the optimal MaxSAT solution to ϕ

```
1  if SAT({C_i | w_i = ∞}) = (False, _) then
2      return ∞
3  cost ← 0
4  w_max ← max{w_i | (C_i, w_i) ∈ φ and w_i < w_max}
5  while True do
6      (state, φ_C) ← SAT({C_i | (C_i, w_i) ∈ φ and w_i ≥ w_max})
7      if state = True and w_max = 0 then
8          return cost
9      else
10         if state = True then
11             w_nax = max{w_i | (C_i, w_i) ∈ φ and w_i < w_max}
12         else
13             BV ← ∅ w_min ← min{w_i | C_i ∈ φ_C and w_i ≠ ∞}
               foreach C_i ∈ φ_C do
14                 if w_i ≠ ∞ then
15                     Let b be a new variable
                       φ ← φ \ {(C_i, w_i)} ∪ {(C_i, w_i - w_min), (C_i ∨ b, w_min)}
16                     BV ← BV ∪ {b}
17             φ ← φ ∪ {(C, ∞) | C ∈ CNF(Σ_{b∈BV} b = 1)} cost ← cost + w_min
```

Algorithm 8 overcomes this problem by utilizing a stratified approach. The aim is to restrict the clauses sent to the SAT solver to force it to concentrate on those with higher weights, which leads the SAT solver to return unsatisfiable cores with clauses having larger weights. Cores with clauses having larger weight are better because they contribute to increasing the cost faster. Clauses with lower weights are used after the SAT solver returns $True$. The algorithm starts by initializing w_{max} to the largest weight smaller than ∞, then in line 6 only the clauses having weight greater than or equal to w_{max} are sent to the SAT solver. The algorithm terminates if the SAT solver returns $True$ and w_{max} is zero (lines 7–8), but if w_{max} is not zero and the formula is satisfiable then w_{max} is decreased to the largest weight smaller than w_{max} (lines 10–11). When the SAT solver returns $False$, the algorithm proceeds as the regular WPM1.

A potential problem with the stratified approach is that in the worst case the algorithm could use more calls to the SAT solver than the regular WPM1. This is because there is no contribution made to the cost when the SAT solver returns $True$ and at the same time $w_{max} > 0$. The authors apply the *diversity heuristic* which decreases w_{max} faster when there is a big variety of distinct weights and assigns w_{max} to the next value of w_i when there is a low diversity among the weights.

4.4 WPM2

In 2007, Marques-Silva and Planes [16] discussed important properties of Fu and Malik that were not mentioned in [9]. If m is the number of clauses in the input formula, they proved that the algorithm performs $O(m)$ iterations and the number of relaxation variables used in the worst case is $O(m^2)$. Marques-Silva and Planes also tried to improve the work of Fu and Malik. Fu and Malik use the pairwise encoding [10] for the constraints on the relaxation variables, which use a quadratic number of clauses. This becomes impractical when solving real-world instances. Instead, Marques-Silva and Planes suggested several other encodings all of which are linear in the number of variables in the constraint [8,10,22].

Another drawback of Fu and Malik is that there can be several blocking variables associated with a given clause. This is due to the fact that a clause C can participate in more than one unsatisfiable core. Each time C is a part of a computed unsatisfiable core, a new blocking variable is added to C. Although the number of blocking variables per clause is possibly large (but still linear), at most one of these variables can be used to prevent the clause from participating in an unsatisfiable core. A simple solution to reduce the search space associated with blocking variables is to require that at most one of the blocking variables belonging to a given clause can be assigned $True$. For a clause C_i, let $b_{i,j}, (1 \leq j \leq t_i)$ be the blocking variables associated with C_i. The condition $\sum_{j=1}^{t_i} b_{i,j} \leq 1$ assures that at most one of the blocking variables of C_i is assigned $True$. This is useful when executing a large number of iterations, and many clauses are involved in a significant number of unsatisfiable cores. The resulting algorithm that incorporated these improvements is called MSU2.

Ansótegui, Bonet and Levy also developed an algorithm for MaxSAT in 2010, called WPM2 [4], where every soft clause C_i is extended with a unique fresh blocking variable b_i. Note that a SAT solver will assign b_i $True$ if C_i is $False$. At every iteration, the algorithm modifies two sets of at-most and at-least constraints on the blocking variables, called AL and AM respectively. The algorithm relies of the notion of *covers*.

Definition 4.4 (Cover)**.** *Given a set of cores L, its set of covers $Covers(L)$ is defined as the minimal partition of $\{1, \ldots, m\}$ such that for every $A \in L$ and $B \in Covers(L)$, if $A \cap B \neq \emptyset$, then $A \subseteq B$.*

The constraints in AL give lower bounds on the optimal cost of ϕ, while the ones in AM ensure that all solutions of the set $AM \cup AL$ are the solutions of AL of minimal cost. This in turn ensures that any solution of $\phi^e \cup CNF(AL \cup AM)$ (if there is any) is an optimal assignment of ϕ.

The authors use the following definition of *cores* and introduced a new notion called *covers* to show how AM is computed given AL.

Algorithm 9. WPM2(ϕ) The WPM2 algorithm for MaxSAT

Input: A MaxSAT instance
$\phi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$
Output: The optimal MaxSAT solution to ϕ

if $SAT(\{C_i \in \phi \mid w_i = \infty\}) = (False, \_)$ **then**
  return ∞
$\phi^e \leftarrow \{C_1 \vee b_1, \ldots, C_m \vee b_m, C_{m+1}, \ldots, C_{m+m'}\}$
$Covers \leftarrow \{\{1\}, \ldots, \{m\}\}$
$AL \leftarrow \emptyset$
$AM \leftarrow \{w_1 b_1 \leq 0, \ldots, w_m b_m \leq 0\}$
while $True$ **do**
  $(state, \phi_C, I) \leftarrow SAT\,(\phi^e \cup CNF(AL \cup AM))$
  if $state = True$ **then**
    return I
  Remove the hard clauses from ϕ_C
  if $\phi_C = \emptyset$ **then**
    return $\emptyset$
  $A \leftarrow \emptyset$
  foreach $C_i \vee b_i \in \phi_C$ **do**
    $A \leftarrow A \cup \{i\}$
  $RC \leftarrow \{B \in Covers \mid B \cap A \neq \emptyset\}$
  $B \leftarrow \bigcup_{B' \in RC} B'$
  $k \leftarrow NewBound(AL, B)$
  $Covers \leftarrow Covers \setminus RC \cup B$
  $AL \leftarrow AL \cup \{\sum_{i \in B} w_i b_i \geq k\}$
  $AM \leftarrow AM \setminus \{\sum_{i \in B'} w_i b_i \leq k' \mid B' \in RC\} \cup \{\sum_{i \in B} w_i b_i \leq k\}$

Definition 4.5 (Core). *A core is a set of indices A such that* $\left(\sum_{i\in A} w_i b_i \geq k\right) \in AL$. *The function* $Core\left(\sum_{i\in A} w_i b_i \geq k\right)$ *returns the core A, and* $Cores(AL)$ *returns* $\{Core(al) \mid al \in AL\}$.

Definition 4.6 (Disjoint cores). *Let* $U = \{U_1, \ldots, U_k\}$ *be a set of unsatisfiable cores, each with a set of blocking variables* $B_i, (1 \leq i \leq k)$. *A core* $U_i \in U$ *is disjoint if for all* $U_j \in U$ *we have* $(R_i \cap R_j = \emptyset$ *and* $i \neq j)$.

Given a set of AL constraints, AM is the set of at-most constraints $\sum_{i\in A} w_i b_i \leq k$ such that $A \in Cover(Cores(AL))$ and k is the solution minimizing $\sum_{i\in A} w_i b_i$ subject to AL and $b_i \in \{True, False\}$. At the beginning, $AL = \{w_1 b_1 \geq 0, \ldots, w_m b_m \geq 0\}$ and the corresponding $AM = \{w_1 b_1 \leq 0, \ldots, w_m b_m \leq 0\}$ which ensures that the solution to $AL \cup AM$ is $b_1 = False, \ldots, b_m = False$. At every iteration, when an unsatisfiable core ϕ_C is identified by the SAT solver, the set of indices of soft clauses in ϕ_C $A \subseteq \{1, \ldots, m\}$ is computed, which is also called a core. Next, the set of covers $RC = \{B' \in Covers \mid B' \cap A \neq \emptyset\}$ that intersect with A is computed, as well as their union $B = \bigcup_{B'\in RC} B'$. The new set of covers is $Covers = Covers \setminus RC \cup B$. The set of at-least constraints AL is enlarged by adding a new constraint $\sum_{i\in B} w_i b_i \geq NewBound(AL, B)$, where $NewBound(AL, B)$ correspond to minimize $\sum_{i\in A} w_i b_i$ subject to the set of constraints $\{\sum_{w_i b_i \geq k}\} \cup AL$ where $k = 1 + \sum\{k' \mid \sum_{i\in A'} w_i b_i \leq k' \in AM$ and $A' \subseteq A\}$. Given AL and B, the computation of $NewBound$ can be difficult since it can be reduced to the subset sum problem in the following way: given $\{w_1, \ldots, w_n\}$ and k, minimize $\sum_{j=1}^{n} w_j x_j$ subject to $\sum_{j=1}^{n} w_j x_j > k$ and $x_j \in \{0, 1\}$. This is equivalent to $NewBound(AL, B)$, where the weights are w_j, $B = \{1, \ldots, n\}$ and $AL = \{\sum_{j=1}^{n} w_j x_j \geq k\}$. In the authors' implementation, $NewBound$ is computed by Algorithm 10.

Algorithm 10. NewBound(AL, B)

```
1 k ← ∑{k' | ∑_{i∈B'} w_i b_i ≤ k' ∈ AM and B' ⊆ B}
2 repeat
3 |   k ← SubsetSum({w_i | i ∈ B}, k)
4 until SAT(CNF(AL ∪ {∑_{i∈B} w_i b_i = k}))
5 return k
```

The $SubsetSum$ function (called in line 3) is an optimization version of the decision subset sum problem. It returns the largest integer $d \leq k$ such that there is a subset of $\{w_i \mid i \in B\}$ that sums to d. To sum up, the WPM2 algorithm groups the identified cores in covers, which are a decomposition of the cores into disjoint sets. Constraints are added so that the relaxation variables in each cover relax a particular weight of clauses k, which is changed to the next largest value the weights of the clauses can sum up to. Computing the next k can be expensive since it relies on the subset sum problem, which is NP-hard.

In [3], Ansótegui *et al.* invented three improvements to WPM2. First, they applied the stratification technique [2]. Second, they introduced a new criteria to decide when soft clauses can be hardened. Finally, they showed that by focusing search on solving to optimality subformulae of the original MaxSAT instance, they efficiency of WPM2 is increased. This allows to combine the strength of exploiting the information extracted from unsatisfiable cores and other optimization approaches. By solving these smaller optimization problems the authors obtained the most significant boost in their new WPM2 version.

4.5 WMSU4

WMSU4 [17] (Algorithm 11) adds at most one blocking variable to each soft clause. Thought, it maintains an upper bound (UB) as well as a lower bound (LB). If the current working formula is satisfiable (line 9), UB is changed to the sum of the weights of the falsified clauses by the solution (I) returned from the SAT solver. On the other hand, if the working formula is unsatisfiable, the SAT solver returns an unsatisfiable core, and the algorithm adds a blocking variable to each clause that has not yet been relaxed in that core. If all the soft clauses in the unsatisfiable core have been relaxed (line 16), then the algorithm updates the lower bound (line 17) and exists the main loop. The following example illustrates how the algorithm works.

Algorithm 11. WMSU4(ϕ) The WMSU4 algorithm for MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: The cost of the optimal MaxSAT solution to ϕ

```
1  if SAT({C | (C, ∞) ∈ φ_H}) = False then
2      return ∞
3  B ← ∅ φ_W ← φ LB ← −1 UB ← 1 + Σ_{i=1}^{|φ_S|} w_i while UB > LB + 1 do
4      (state, φ_C, I) ← SAT({C | (C, w) ∈ φ_W} ∪ CNF(Σ_{b_i∈B} w_i b_i ≤ UB − 1))
5      if state = True then
6          UB ← Σ_{b_i∈B} w_i(1 − I(C_i \ b_i))
7      else
8          foreach (C_i, w_i) ∈ φ_C ∩ φ_S do
9              if w ≠ ∞ then
10                 B' ← B' ∪ {b_i}
11                 φ_W ← φ_W \ {(C_i, w_i)} ∪ {(C_i ∨ b_i, w_i)}
12         if B' = ∅ then
13             LB ← UB − 1
14         else
15             B ← B ∪ B'
16             LB ← UpdateBound({w_i | b_i ∈ B}, LB)
17 return UB
```

5 Core-Guided Binary Search Algorithms

Core-guided binary search algorithms are similar to binary search algorithms except that they do not augment all the soft clauses with blocking variables before the beginning of the main loop. Heras, Morgado and Marques-Silva proposed this technique in [11] (see Algorithm 12).

Algorithm 12. CoreGuided-BS(ϕ) Core-guided binary search algorithm for solving MaxSAT.

Input: A MaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: The cost of the optimal MaxSAT solution to ϕ

```
1  state ← SAT({C_i | (C_i, ∞) ∈ φ_H})
2  if state = False then
3      return ∅
4  φ_W ← φ
5  LB ← −1
6  UB ← 1 + Σ_{i=1}^{|φ_S|} w_i
7  B ← ∅
8  while LB + 1 < UB do
9      mid ← ⌊(LB+UB)/2⌋
10     (state, φ_C, I) ← SAT({C | (C, w) ∈
       φ_W} ∪ CNF(Σ_{b_i∈B} w_i b_i ≤ mid))
11     if state = True then
12         UB ← Σ_{i=1}^{|φ_S|} w_i(1 − I(C_i \ {b_i}))
13         lastI ← I
14     else
15         if φ_C ∩ φ_S = ∅ then
16             LB ← UpdateBound({w_i | b_i ∈ B}, mid) − 1
17         else
18             foreach (C_i, w_i) ∈ φ_C ∩ φ_S do
19                 let b_i be a new blocking variable
20                 B ← B ∪ {b_i}
21                 φ_W ← φ_W \ {(C_i, w_i)} ∪ {(C_i ∨ b_i, w_i)}
22     return lastI
```

Similar to other algorithms, CoreGuided-BS begins by checking the satisfiability of the hard clauses (lines 1–3). Then it initializes the lower bound (line 4), the upper bound (line 5) and the set of blocking variables (line 6) respectively to -1, one plus the sum of the weights of the soft clauses and $\emptyset$. At each iteration of the main loop (lines 7–21) a SAT solver is called on the working formula with a constraint ensuring that the sum of the weights of the relaxed soft clauses is less than or equal the middle value (line 9). If the formula is satisfiable (line 10),

Algorithm 13. DisjointCoreGuided-BS(ϕ) Core-guided binary search extended with disjoint cores for solving MaxSAT.

```
   Input: A MaxSAT instance φ = φ_S ∪ φ_H
   Output: A MaxSAT solution to φ
1  if SAT({C | (C, ∞) ∈ φ_H}) = False then
2      return ∅
3  φ_W ← φ
4  C ← ∅
5  repeat
6      foreach C_i ∈ C do
7          if LB_i + 1 = UB_i then
8              mid_i ← UB_i
9          else
10             mid_i ← ⌊(LB_i + UB_i)/2⌋
11     (state, φ_C, I) ← SAT({C | (C, w) ∈
       φ_W} ∪ ⋃_{C_i∈C} CNF(Σ_{b_i∈B} w_i b_i ≤ mid_i))
12     if state = True then
13         lastI ← I
14         foreach C_i ∈ C do
15             UB_i ← Σ_{b_r∈B} w_r(1 − I(C_r \ {b_r})))
16     else
17         subC ← IntersectingCores(φ_C, C)
18         if φ_C ∩ φ_S = ∅ and |subC| = 1 then
19             LB ← mid
20         else
21             foreach (C_i, w_i) ∈ φ_C ∩ φ_S do
22                 let b_i be a new blocking variable
23                 B ← B ∪ {b_i}
24                 φ_W ← φ_W \ {(C_i, w_i)} ∪ {(C_i ∨ b_i, w_i)}
25             LB ← 0
26             UB ← 1 + Σ_{b_i∈B} w_i
27             foreach (B_i, LB_i, mid_i, UB_i) ∈ subC do
28                 B ← B ∪ B_i
29                 LB ← LB + LB_i
30                 UB ← UB + UB_i
31             C ← C \ subC ∪ {(B, LB, 0, UB)}
32 until ∀_{C_i∈C} UB_i ≤ LB_i + 1
33 return lastI
```

the upper bound is updated to the sum of the falsified soft clauses by the current assignment (line 11). Otherwise, if all the soft clauses have been relaxed (line 14), then the lower bound is updated (line 15), and if not, non-relaxed sot clauses

belonging to the core are relaxed (lines 17–19). The main loop continues as long as $LB + 1 < UB$.

The core-guided binary search approach was improved by Heras [11] *et al.* with disjoint cores (see Definition 4.6).

Core-guided binary search methods with disjoint unsatisfiable cores maintain smaller lower and upper bounds for each disjoint core instead of just one global lower bound and one global upper bound. Thus, the algorithm will add multiple smaller cardinality constraints on the sum of the weights of the soft clauses rather than just one global constraint.

To maintain the smaller constraints, the algorithm keep information about the previous cores in a set called $\mathcal{C}$ initialized to $\emptyset$ (line 4) before the main loop. Whenever the SAT solver returns $False$ (line 12) it also provides a new core and a new entry $C_i = (B_i, LB_i, mid_i, UB_i)$ is added in $\mathcal{C}$ for U_i, where B_i is the set of blocking variables associated with the soft clauses in U_i, LB_i is a lower bound, mid_i is the current middle value and UB_i is an upper bound. The main loop terminates when for each $C_i \in \mathcal{C}$, $LB_i + 1 \geq UB_i$ (line 33). For each entry in $\mathcal{C}$, its middle value is calculated (lines 6–10) and a constraint for each entry is added to the working formula before calling the SAT solver on it (line 11). If the working formula is unsatisfiable (line 16), then, using $IntersectingCores$, every core that intersects the current core is identified and its corresponding entry is added to $sub\mathcal{C}$ (line 17). If the core does not contain soft clauses that need to be relaxed and $|sub\mathcal{C}| = 1$ (line 18), then LB is assigned the value of the midpoint (line 19). On the other hand, if there exists clauses that has not been relaxed yet then the algorithm relaxes them (lines 21–24) and a new entry for the current core is added to $\mathcal{C}$ which accumulates the information of the previous cores in $sub\mathcal{C}$ (lines 25–31).

SAT-based MaxSAT solvers rely heavily on the hardness of the SAT formulae returned by the underlying SAT solver used. Obviously, the location of the optimum solution depends on the structure of the instances returned and the number of iterations it takes to switch from $True$ to $False$ (or from $False$ to $True$).

6 Portfolio MaxSAT Techniques

The results of the MaxSAT Evaluations suggest there is no absolute best algorithm for solving MaxSAT. This is because the most efficient solver often depends on the type of instance. Having an oracle able to predict the most suitable MaxSAT solver for a given instance would result in the most robust solver. The success of SATzilla for SAT was due to a regression function which was trained to predict the performance of every solver in the given set of solvers based on the features of an instance. When faced with a new instance, the solver with the best predicted runtime is run on the given instance. The resulting SAT portfolios excelled in the SAT Competitions in 2007 and in 2009 and pushed the state-of-the-art in SAT solving. When this approach is extended to (WP)MaxSAT, the resulting portfolio can achieve significant performance improvements on a representative set of instances.

ISAC [6] (Instance-Specific Algorithm Configuration) is one of the most successful MaxSAT portfolio algorithms. It works by computing a representative feature vector that characterizes the given input instance in order to identify clusters of similar instances. The data is therefore clustered into non-overlapping groups and a single solver is selected for each group based on some performance characteristic. Given a new instance, its features are computed and it is assigned to the nearest cluster. The instance is then solved by the solver assigned to that cluster.

7 Experimental Investigation

This experimental investigation has been done of the following solvers:

(1) **WMiFuMax** is an unsatisfiability-based WPMaxSAT solver based on the technique of Fu and Malik. It which works by identifying unsatisfiable subformulae. MiFuMax placed third in the WPMaxSAT industrial category of the 2013 MaxSAT evaluation. The SAT solver used is called MiniSAT. Author: Mikoláš Janota.
(2) **QWMaxSAT** is a weighted version of QMaxSAT [12] and is available freely online. This solver is a satisfiability-based solver built on top of version 2.0 of MiniSAT.
(3) **Sat4j** [13] is a satisfiability-based WPMaxSAT solver. The solver works by translating WPMaxSAT instances into pseudo-Boolean optimization ones. Sat4j avoids adding blocking variables to both hard and unit clauses.
(4) **MSUnCore** [15] is an unsatisfiability-based WPMaxSAT solver built on top the SAT solver PicoSAT. Clauses in identified cores are then relaxed by adding a relaxation variable to each clause. Cardinality constraints are encoded using several encodings, such as the pairwise and bitwise encodings [20,21].
(5) **Maxsatz2013f** is a branch and bound solver that placed first in the WPMaxSAT random category of the 2013 MaxSAT evaluation. It is based on an earlier solver called Maxsatz [14], which incorporates the technique developed for the famous SAT solver Satz. At each node, it transforms the instance into an equivalent one by applying efficient refinements of unit resolution ($(A \vee B)$ and $(\neg B)$ yield A) which replaces $\{(x), (y), (\neg x \vee \neg y)\}$ with $\{\square, (x \vee y)\}$ and $\{(x), (\neg x \vee y), (\neg x \vee z), (\neg y \vee \neg z)\}$ with $\{\square, (\neg x \vee y \vee z), (x \vee \neg y \vee \neg z)\}$, where $\square$ is the empty clause. It implements a lower bound method (enhanced with failed literal detection) that increments the lower bound by one for every disjoint inconsistent subset that is detected by unit propagation.
(6) **WMaxSatz-2009** and **WMaxSatz+** are branch and bound solvers that use transformation rules [14] which can be implemented efficiently as a by-product of unit propagation or failed literal detection. This means that the transformation rules can be applied at each node of the search tree. Authors: Josep Argelich, Chu Min Li, Jordi Planes and Felip Manyà.

ISAC+ [6] (Instance-Specific Algorithm Configuration) is a portfolio of algorithm which, given a WPMaxSAT instance, selects the solver better suited for that instance. A regression function is trained to predict the performance of every solver in the given set of solvers based on the features of an instance. When faced with a new instance, the solver with the best predicted runtime is run on the given instance. ISAC+ uses a number of branch and bound solvers as well as SAT-based, including QMaxSAT, WMaxSatz-2009 and WMaxSatz+. Authors: Carlos Ansótegui, Joel Gabas, Yuri Malitsky and Meinolf Sellmann.

The solvers were run on a machine with an Intel® Core™i5 CPU clocked at 2.4 GHz with 5.7 GB of RAM running elementary OS Linux and a timeout was set to 1000 s. We picked elementary OS because it does not consume too many resources to run and thus giving enough room for the solvers to run. The benchmarks we used are the WPMaxSAT instances (http://www.maxsat.udl.cat/13/benchmarks/index.html) of the 2013 MaxSAT Evaluation and are divided into three categories: (1) random, (2) crafted and (3) industrial. For each category, we present the constituting sets of instances and their sizes, the number of instances solved by each solver and the amount of time it took each solver to work on each set of instances.

7.1 Random Category

The three sets of instances in the random category are mentioned in Table 1. Tables 2 and 3 summarize the results of the random benchmarks.

Table 1. Benchmark instances of the Random category

Name	Abbreviation	# of instances
wpmax2sat-lo	lo	30
wpmax2sat-me	me	30
wpmax2sat-hi	hi	30
wpmax3sat-hi	3hi	30

The branch and bound solvers MaxSatz2013f, WMaxSatz-2009 and WMaxSatz+ performed considerably better than the SAT-based solvers in the random category. In particular, MaxSatz2013f finished the four benchmarks under 16 min, while WMiFuMax, MSUnCore and Sat4j timedout on most instances. MaxSatz2013f placed first in the random category in the 2013 MaxSAT Evaluation. The top non branch and bound solver is ISAC+, which placed third in the random category in 2014 (Fig. 1).

Table 2. Number of instances solved in the random category

Solver	lo	me	hi	3hi
MiFuMax	0	0	0	0
QWMaxSAT	0	0	0	0
Sat4j	0	0	0	0
MSUnCore	0	0	0	0
MaxSatz2013f	30	30	29	30
WMaxSatz-2009	30	30	29	30
WMaxSatz+	30	30	29	30
ISAC+	29	8	1	10

Table 3. Percentages of instances solved in the random category

Solver	lo (%)	me (%)	hi (%)	3hi (%)	Total (%)
WMiFuMax	0	0	0	0	0
QWMaxSAT	0	0	0	0	0
Sat4j	0	0	0	0	0
MSUnCore	0	0	0	0	0
MaxSatz2013f	100	100	96.7	100	99.2
WMaxSatz-2009	100	100	96.7	100	99.2
WMaxSatz+	100	100	96.7	100	99.2
ISAC+	96.7	26.7	3.3	33.3	40

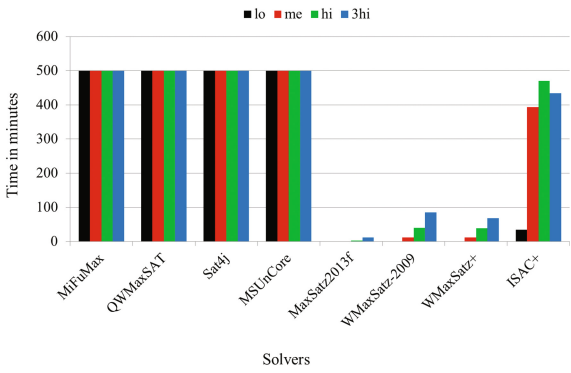

Fig. 1. Time results for the random category

7.2 Crafted Category

The seven sets of instances in the crafted category are mentioned in Table 4. Tables 5, 6 and 7 summarize the results of the crafted benchmarks.

Table 4. Benchmark instances of the Crafted category

Name	Abbreviation	# of instances
Auctions/auc-paths	Auc/paths	86
Auctions/auc-scheduling	Auc/sch	84
CSG	csg	10
Min-enc/planning	Planning	56
Min-enc/warehouses	Warehouses	18
Pseudo/miplib	miplib	12
Random-net	rnd-net	74

Table 5. Number of instances solved by each solver

Solver	Auc/paths	Auc/sch	csg	Planning	Warehouses	miplib	rnd-net
WMiFuMax	84	84	5	23	0	1	8
QWMaxSAT	84	84	10	56	2	4	1
Sat4j	55	55	10	56	1	4	0
MSUnCore	84	84	6	53	0	0	0
MaxSatz2013f	81	81	1	41	6	4	1
WMaxSatz-2009	67	67	1	45	6	3	0
WMaxSatz+	66	66	1	45	6	2	0
ISAC+	84	84	4	53	18	3	55

Table 6. Percentages of instances solved in the crafted category

Solver	Auc/paths (%)	Auc/sch (%)	csg (%)	Planning (%)
WMiFuMax	2.3	100	50	41.1
QWMaxSAT	52.3	100	100	100
Sat4j	31.4	65.5	100	100
MSUnCore	16.3	100	60	94.6
MaxSatz2013f	100	96.4	10	73.2
WMaxSatz-2009	100	79.8	10	80.4
WMaxSatz+	100	78.6	10	80.4
ISAC+	100	100	40	94.6

The following two tables summarize the time results on the benchmarks of the crafted category and the total number of instances solved by each solver in each set.

Table 7. Percentages of instances solved in the crafted category and the total number of instances solved

Solver	Warehouses (%)	miplib (%)	rnd-net (%)	Total (%)
WMiFuMax	0	8.3	10.8	30.1
QWMaxSAT	11.1	33.3	1.4	57
Sat4j	5.6	33.3	0	48
MSUnCore	0	0	0	38.7
MaxSatz2013f	33.3	33.3	1.4	49.7
WMaxSatz-2009	33.3	25	0	47
WMaxSatz+	33.3	16.7	0	45.6
ISAC+	100	25	74.3	76.3

As it can be noticed from the results, ISAC+ is the winner of the crafted category. Indeed, the winner of this category in the 2014 MaxSAT Evaluation is ISAC+, and in the 2013 evaluation it placed second. Generally, SAT-based and branch and bound solvers perform nearly equally on crafted instances (Fig. 2).

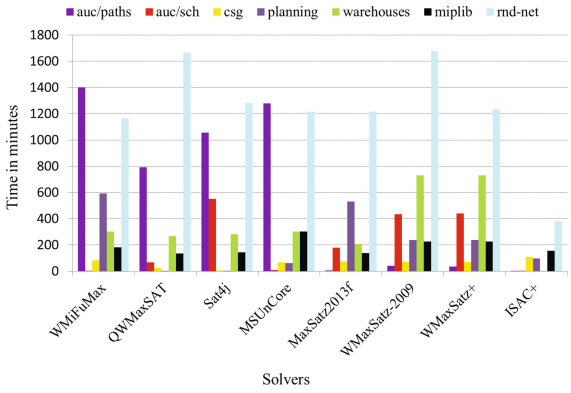

Fig. 2. Time results for the crafted category

7.3 Industrial Category

The seven sets of instances in the industrial category are mentioned in Table 8. Tables 9 and 10 summarize the results of the industrial benchmarks.

It is clear that SAT-based solvers outperform branch and bound ones on industrial instances. The winner solver of this category in the 2013 MaxSAT

Table 8. Benchmark instances of the Industrial category

Name	Abbreviation	# of instances
wcsp/spot5/dir	wcsp-dir	21
wcsp/spot5/log	wcsp-log	21
Haplotyping-pedigrees	HT	100
Upgradeability-problem	UP	100
Preference_planning	PP	29
Packup-wpms	PWPMS	99
Timetabling	TT	26

Table 9. Number of instances solved in the industrial category

Solver	wcsp-dir	wcsp-log	HT	UP	PP	PWPMS	TT
WMiFuMax	6	6	85	100	11	46	0
QWMaxSAT	14	13	20	0	29	17	8
Sat4j	3	3	15	37	28	2	8
MSUnCore	14	14	89	100	25	0	0
MaxSatz2013f	4	4	0	0	5	25	0
WMaxSatz-2009	4	3	0	41	5	12	0
WMaxSatz+	4	3	0	41	5	12	0
ISAC+	17	7	15	100	9	99	9

Table 10. Percentages of instances solved in the industrial category

Solver	wcsp-dir (%)	wcsp-log (%)	HT (%)	UP (%)	PP (%)	PWPMS (%)	TT (%)	Total (%)
WMiFuMax	28.6	28.6	85	100	15.2	46	0	43.3
QWMaxSAT	66.7	61.9	20	0	40	17	30.8	34.5
Sat4j	14.3	14.3	15	37	38.7	2	30.8	21.7
MSUnCore	66.7	66.7	89	100	34.5	0	0	51
MaxSatz2013f	19	19	0	0	6.9	25	0	10
WMaxSatz-2009	19	14.3	0	41	5	12	0	13
WMaxSatz+	19	14.3	0	41	6.9	12	0	13
ISAC+	81	33.3	15	100	12.4	100	34.6	53.8

evaluation is ISAC+ and the same solver placed second in the 2014 evaluation (Fig. 3).

Generally, we can notice that on industrial instances, SAT-based solvers are performed considerably better than branch and bound solvers which performed poorly. On the other hand, branch and bound solvers outperformed SAT-based ones on random instances.

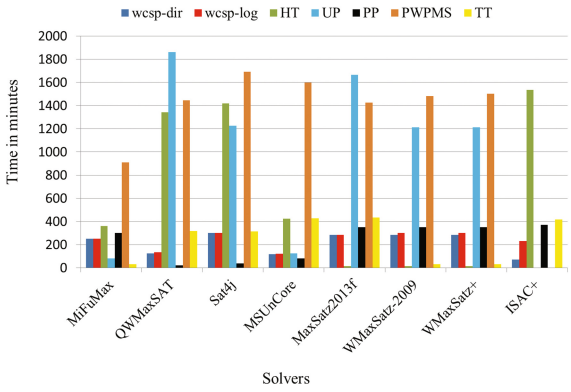

Fig. 3. Time results for the industrial category

8 Conclusion

This paper discusses solving the MaxSAT problem by iteratively calling a SAT solver on the input formula. It consists of two main parts: (1) Describing several SAT-based algorithms which competed in the MaxSAT evaluations. (2) An experimental investigation and comparison between a number of SAT-based solvers and branch-and-bound solvers.

The results show that, in general, SAT-based solvers are best suited for instances arising from practical applications. This is apparent from the time results on the "Industrial" benchmark instances. In addition, SAT-based solvers showed moderate to good results on "Crafted" instances. Branch-and-bound solvers have been found to be efficient for solving "Random" instances and some crafted ones.

Acknowledgments. I would like to thank Dr. Hassan Aly (Department of Mathematics, Cairo University), Dr. Rasha Shaheen (Department of Mathematics, Cairo University) and Dr. Carlos Ansótegui (University of Lleida) for helping me throughout this research.

References

1. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial maxsat through satisfiability testing. In: Theory and Applications of Satisfiability Testing-SAT 2009, pp. 427–440 (2009)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving sat-based weighted maxSAT solvers. In: Principles and Practice of Constraint Programming, pp. 86–101. Springer (2012)
3. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving WPM2 for (weighted) partial maxSAT. In: Principles and Practice of Constraint Programming, pp. 117–132. Springer (2013)

4. Ansótegui, C., Bonet, M.L., Levy, J.: A New Algorithm for Weighted Partial maxSAT (2010)
5. Ansótegui, C., Bonet, M.L., Levy, J.: SAT-based maxSAT algorithms. Artif. Intell. **196**, 77–105 (2013)
6. Ansótegui, C., Malitsky, Y., Sellmann, M.: MaxSAT by Improved Instance-Specific Algorithm Configuration (2014)
7. Davies, J., Bacchus, F.: Postponing optimization to speed up maxSAT solving. In: Principles and Practice of Constraint Programming, pp. 247–262. Springer (2013)
8. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. JSAT **2**(1–4), 1–26 (2006)
9. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Theory and Applications of Satisfiability Testing-SAT 2006, pp. 252–265 (2006)
10. Gent, I.P.: Arc consistency in sat. In: ECAI, vol. 2, pp. 121–125 (2002)
11. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Proceedings of the AAAI National Conference (AAAI) (2011)
12. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: a partial Max-SAT solver system description. J. Satisfiability Boolean Model. Comput. **8**, 95–100 (2012)
13. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2 system description. J. Satisfiability Boolean Model. Comput. **7**, 59–64 (2010)
14. Li, C.M., Manyà, F., Mohamedou, N., Planes, J.: Exploiting cycle structures in Max-SAT. In: Theory and Applications of Satisfiability Testing-SAT 2009, pp. 467–480 (2009)
15. Marques-Silva, J.: The MSUNCORE MaxSAT solver. In: SAT 2009 Competitive Events Booklet: Preliminary Version, p. 151 (2009)
16. Marques-Silva, J., Planes, J.: On Using Unsatisfiability for Solving Maximum Satisfiability. arXiv preprint arXiv:0712.1097 (2007)
17. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 408–413. ACM (2008)
18. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: a survey and assessment. Constraints **18**(4), 478–534 (2013)
19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference, pp. 530–535. ACM (2001)
20. Prestwich, S.: Variable dependency in local search: Prevention is better than cure. In: Theory and Applications of Satisfiability Testing-SAT 2007, pp. 107–120. Springer (2007)
21. Prestwich, S.D.: CNF encodings. In: Handbook of Satisfiability, vol. 185, pp. 75–97 (2009)
22. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinalityconstraints. In: Principles and Practice of Constraint Programming-CP 2005, pp. 827–831 (2005)