

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) problem is the problem of finding a truth assignment that maximizes the number of satisfied clauses of a given Boolean formula in Conjunctive Normal Form (CNF). 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 is 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 the benchmarks of the MaxSAT Evaluations.

Contents

1	Introduction and Preliminaries	4
2	Linear Search Algorithms	5
3	Binary Search-based Algorithms	7
4	Core-guided Algorithms	12
4.1	Fu and Malik's algorithm	13
4.2	WPM1	14
4.3	Improved WPM1	15
4.4	WPM2	17
4.5	WMSU1-ROR	20
4.6	WMSU3	23
4.7	WMSU4	25
5	Core-guided Binary Search Algorithms	26
6	Portfolio MaxSAT Techniques	31
7	Translating Pseudo-Boolean Constraints into CNF	31
7.1	Introduction	31
7.2	Encoding method	32
7.3	Complexity of the encoding	33
7.3.1	Polynomial cases	33
7.3.2	Exponential cases	34
7.4	Other encoding techniques	34
8	Experimental Investigation	35
8.1	Solvers descriptions	35
8.2	Benchmarks descriptions	37
8.3	Results	37
8.3.1	Random category	37
8.3.2	Crafted category	39
8.3.3	Industrial category	40
9	Acknowledgments	42

List of Algorithms

1	LinearUNSAT(ϕ) Linear search UNSAT-based algorithm for solving WP-	
	MaxSAT.	5
2	LinearSAT(ϕ) Linear search SAT-based algorithm for solving WPMa	6
3	BinS-WPMa	8
4	BinLin-WPMa	10
5	BitBased-WPMa	11
6	Fu&Malik(ϕ) Fu and Malik's algorithm for solving P	13
7	WPM1(ϕ) The WPM1 algorithm for WPMa	14
8	ImprovedWPM1(ϕ) The stratified approach for WPM1 algorithm.	16
9	WPM2(ϕ) The WPM2 algorithm for WPMa	18
10	NewBound(AL, B)	19
11	WMSU1-ROR(ϕ)	22
12	Hard($(C_i, w_i), R$) Determines if a clause is hard or not	23
13	ROR($(C_i, w_i), R$) Determines if a clause is hard or not or if its ancestors are used at most once	23
14	WMSU3(ϕ) The WMSU3 algorithm for WPMa	24
15	WMSU4(ϕ) The WMSU4 algorithm for WPMa	25
16	CoreGuided-BS(ϕ) Core-guided binary search algorithm for solving WP-	
	MaxSAT.	27
17	DisjointCoreGuided-BS(ϕ) Core-guided binary search extended with disjoint cores for solving WPMa	29

1 Introduction and Preliminaries

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 $\{C_1, C_2, \dots, C_k\}$ 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 satisfiability problem (SAT) is deciding whether ϕ has a satisfying truth assignment (i.e., an assignment to the variables of ϕ that satisfies every clause). The *Maximum Satisfiability* (MaxSAT) problem asks for a truth assignment that maximizes the number of satisfied clauses in ϕ .

Many theoretical and practical problems can be encoded into SAT and MaxSAT such as debugging [51], circuits design and scheduling of how an observation satellite captures photos of Earth [56], course timetabling [11, 45, 41, 34], software package upgrades [24], routing [58, 46], reasoning [52] and protein structure alignment in bioinformatics [50].

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

In general, exact MaxSAT solvers follow one of two approaches: successively calling a SAT solver (sometimes called the SAT-based approach) and the branch and bound approach. The former converts each MaxSAT problem with different hypothesized maximum weights into multiple SAT problems and uses a SAT solver to solve these SAT problems to determine the actual solution. The SAT-based approach converts the WPMMaxSAT problem into a sequence of SAT instances which can be solved using SAT solvers. One way to do this, given an unweighted MaxSAT instance, is to check if there is an assignment that 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 [43, 8].

The second approach utilizes a depth-first branch and bound search in the space of possible assignments. An evaluation function which computes a bound is applied at each search node to determine any pruning opportunity. This paper surveys the satisfiability-based approach and provides an experimental investigation and comparison between the performances of both approaches on sets of benchmarks.

Because of the numerous calls to a SAT solver this approach makes, any improvement to SAT algorithms immediately benefits MaxSAT SAT-based methods. Experimental results from the MaxSAT Evaluations¹ have shown that SAT-based solvers are more competent to handle large MaxSAT instances from industrial applications than branch and bound methods.

¹Web page: <http://www.maxsat.udl.cat>

2 Linear Search Algorithms

A simple way to solve WPMaXSAT 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* (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.

Algorithm 1 employs the first method to search for the optimal cost by maintaining (maintaining a lower bound initialized to 0) (line 1).

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

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

```

1  $LB \leftarrow 0$ 
2 foreach  $(C_i, w_i) \in \phi_S$  do
3   | let  $b_i$  be a new blocking variable
4   |  $\phi_S \leftarrow \phi_S \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
5 while True do
6   |  $(state, I) \leftarrow SAT(\{C \mid (C, w) \in \phi\} \cup CNF(\sum_{i=1}^{|\phi_S|} w_i b_i \leq LB))$ 
7   | if  $state = True$  then
8   |   | return  $I$ 
9   |  $LB \leftarrow UpdateBound(\{w \mid (C, w) \in \phi_S\}, LB)$ 

```

Next, the algorithm relaxes each soft clause with a new variable in lines 2-4. The formula ϕ now contains each soft clause augmented with a new blocking variable. The while loop in lines 5-9 sends the clauses of ϕ (without the weights) to a SAT solver (line 6). If the SAT solver returns *True*, then LinearUNSAT terminates returning a solution (lines 7-8). Otherwise, the lower bound is updated and the loop continues until the SAT solver returns *True*. The function *UpdateBound* in line 9 updates the lower bound either by simply increasing it or by other means that depend on the distribution of the weights of the input formula. Later in this paper we will see how the subset sum problem can be a possible implementation of *UpdateBound*. Note that it could be inefficient if *UpdateBound* changes LB by one in each iteration. Consider a WPMaXSAT formula with five soft clauses having the weights 1, 1, 1, 1 and 100. The cost of the optimal solution can not be anything else other than 0, 1, 2, 3, 4, 100, 101, 102, 103 and 104. Thus, assigning LB any of the values 5, ..., 99 is unnecessary and will result in a large number of iterations.

Example 2.1. Let $\phi = \phi_S \cup \phi_H$, where $\phi_S = \{(x_1, 5), (x_2, 5), (x_3, 10), (x_4, 5), (x_5, 10), (x_6, 5), (\neg x_6, 10)\}$ and $\phi_H = \{\neg x_1 \vee \neg x_2, \infty, (\neg x_2 \vee \neg x_3, \infty), (\neg x_3 \vee \neg x_4, \infty), (\neg x_4 \vee \neg x_5, \infty), (\neg x_5 \vee \neg x_1, \infty)\}$. If we run *LinearUNSAT* on ϕ , the soft clauses will be relaxed $\{(x_1 \vee b_1, 5), (x_2 \vee b_2, 5), (x_3 \vee b_3, 10), (x_4 \vee b_4, 5), (x_5 \vee b_5, 10), (x_6 \vee b_6, 5), (\neg x_6 \vee b_7, 10)\}$ and LB is initialized to 0. The sequence of iterations are

1. The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 0)$ is included, state = False, $LB = 5$.
2. The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 5)$ is included, state = False, $LB = 10$.
3. The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 10)$ is included, state = False, $LB = 15$.
4. The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 15)$ is included, state = False, $LB = 20$.
5. The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 20)$ is included, state = True. The SAT solver returns the assignment $I = \{x_1 = \text{False}, x_2 = \text{False}, x_3 = \text{True}, x_4 = \text{False}, x_5 = \text{True}, x_6 = \text{False}, b_1 = \text{True}, b_2 = \text{True}, b_3 = \text{False}, b_4 = \text{True}, b_5 = \text{False}, b_6 = \text{True}, b_7 = \text{False}\}$, which leads to a *WPMaXSAT* solution if we ignore the values of the $b_i, (1 \leq i \leq 7)$ variables with cost 20.

The next algorithm describes the SAT-based technique. Algorithm 2 starts by initializing the upper bound to one plus the the sum of the weights of the soft clauses (line 1).

Algorithm 2: LinearSAT(ϕ) Linear search SAT-based algorithm for solving WP-MaXSAT.

Input: A WPMaXSAT instance $\phi = \phi_S \cup \phi_H$
Output: A WPMaXSAT 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 \perp 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 \perp **return** lastI
- 8 lastI $\leftarrow I$
- 9 $UB \leftarrow \sum_{i=1}^{|\phi_S|} w_i (1 - I(C_i \setminus \{b_i\}))$

In each iteration of algorithm 2 except the last, the formula is satisfiable. The cost of the optimal solution is found immediately after the transition from satisfiable to unsatisfiable instance. LinearSAT begins by initializing the upper bound to one plus the sum of the weights of the soft clauses (line 1). The while loop (lines 4-8) continues until the formula becomes unsatisfiable (line 6), then the algorithm returns a WPMaXSAT solution and terminates (line 7). As long as the formula is satisfiable, the formula is sent to the SAT

solver along with the constraint assuring that the sum of the weights of the falsified soft clauses is less than $UB - 1$ (line 5), and the upper bound is updated to the sum of the weights of the soft clauses falsified by the assignment returned by the SAT solver (line 8).

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 it uses less iterations and thus the problem is solved with less SAT calls.

Example 2.2. *If we run LinearSAT on ϕ from the previous example, the soft clauses will be relaxed $\{(x_1 \vee b_1, 5), (x_2 \vee b_2, 5), (x_3 \vee b_3, 10), (x_4 \vee b_4, 5), (x_5 \vee b_5, 10), (x_6 \vee b_6, 5), (\neg x_6 \vee b_7, 10)\}$ and UB is initialized to $1 + (5 + 5 + 5 + 5 + 10 + 10 + 10) = 51$. The sequence of iterations are*

1. *The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 50)$ is included, $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = False, x_4 = False, x_5 = False, x_6 = False, b_1 = True, b_2 = True, b_3 = True, b_4 = True, b_5 = True, b_6 = True, b_7 = False\}$, $UB = 5 + 5 + 10 + 5 + 10 + 5 = 40$.*
2. *The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 40 - 1)$ is included, $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = False, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = True, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$, $UB = 5 + 5 + 10 + 5 + 5 = 30$.*
3. *The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 30 - 1)$ is included, $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = False, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$, $UB = 5 + 5 + 5 + 5 = 20$.*
4. *The constraint $CNF(5b_1 + 5b_2 + 10b_3 + 5b_4 + 10b_5 + 5b_6 + 10b_7 \leq 20 - 1)$ is included, $state = False$. The assignment from the previous step is indeed a solution to ϕ if we ignore the values of the $b_i, (1 \leq i \leq 7)$ variables with cost 20.*

3 Binary Search-based Algorithms

The number of iterations linear search algorithms for WPMaXSAT can take is linear in the sum of the weights of the soft clauses. Thus, in the worst case the a linear search WPMaXSAT algorithm 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.

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

Input: A WPMaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A WPMaxSAT 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  $LB \leftarrow -1$ 
5  $UB \leftarrow 1 + \sum_{i=1}^{|\phi_S|} w_i$ 
6 foreach  $(C_i, w_i) \in \phi_S$  do
7   let  $b_i$  be a new blocking variable
8    $\phi_S \leftarrow \phi_S \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
9 while  $LB + 1 < UB$  do
10   $mid \leftarrow \lfloor \frac{LB+UB}{2} \rfloor$ 
11   $(state, I) \leftarrow SAT(\{C \mid (C, w) \in \phi\} \cup CNF(\sum_{i=1}^{|\phi_S|} w_i b_i \leq mid))$ 
12  if  $state = True$  then
13     $lastI \leftarrow I$ 
14     $UB \leftarrow \sum_{i=1}^{|\phi_S|} w_i (1 - I(C_i \setminus \{b_i\}))$ 
15  else
16     $LB \leftarrow UpdateBound(\{w_i \mid 1 \leq i \leq |\phi_S|\}, mid) - 1$ 
17 return  $lastI$ 

```

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

In the following example, *UpdateBound* assigns $mid + 1$ to LB .

Example 3.1. Consider ϕ in example 2.1 with all the weights of the soft clauses set to 1. At the beginning, $LB = -1$, $UB = 8$. The following are the sequence of iterations algorithm 3 executes.

1. $mid = \lfloor \frac{8+(-1)}{2} \rfloor = 3$, the constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 \leq 3)$ is included, $state = False$, $LB = 3$, $UB = 8$.
2. $mid = \lfloor \frac{8+3}{2} \rfloor = 5$, the constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 \leq 5)$ is

included, state = True, $I = \{x_1 = \text{False}, x_2 = \text{False}, x_3 = \text{True}, x_4 = \text{False}, x_5 = \text{True}, x_6 = \text{False}, b_1 = \text{True}, b_2 = \text{True}, b_3 = \text{False}, b_4 = \text{True}, b_5 = \text{False}, b_6 = \text{True}, b_7 = \text{False}\}$, $UB = 4$, $LB = 3$. The assignment I is indeed an optimal one, falsifying four clauses.

It is often stated that a binary search algorithm performs better than linear search. Although this is true most of the time, there are instances for which linear search is faster than binary search. Let k be the sum of the soft clauses falsified by the assignment returned by the SAT solver in the first iteration. If k is indeed the optimal solution, linear search methods would discover this fact in the next iteration, while binary search ones would take $\log k$ iterations to declare k as the optimal cost. In order to benefit from both search methods, An *et al.*[3] developed a PMaxSAT algorithm called QMaxSAT (version 0.4) that alternates between linear search and binary search (see algorithm 4).

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

Input: A WPMaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A WPMaxSAT 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  $lastI$ 

```

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-WPMaxSAT 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-WPMaxSAT 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 is found.

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

Input: A WPMaxSAT instance $\phi = \phi_S \cup \phi_H$
Output: A WPMaxSAT 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  $k \leftarrow \lceil \lg(\sum_{i=1}^{|\phi_S|} w_i) \rceil$ 
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  $s_0, \dots, s_k \in \{0, 1\}$  be constants such that
15        $\sum_{i=1}^{|\phi_S|} w_i (1 - I(C_i \setminus \{b_i\})) = \sum_{j=0}^k 2^j s_j$  //  $s_0, \dots, s_k$  are the binary
16       representation of the current cost
17      $CurrBit \leftarrow \max(\{j \mid j < CurrBit \text{ and } s_j = 1\} \cup \{-1\})$ 
18     if  $CurrBit \geq 0$  then
19        $cost \leftarrow \sum_{j=CurrBit}^k 2^j s_j$ 
18   else
19      $CurrBit \leftarrow CurrBit - 1$ 
20      $cost \leftarrow cost + 2^{CurrBit}$ 
21 return  $lastI$ 

```

At the beginning of the algorithm as in the previous ones, the satisfiability of the hard clauses are 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 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).

Example 3.2. Consider ϕ from example 2.1 with all the weights of the soft clauses being 1. At the beginning of the algorithm, the soft clauses are relaxed and the formula becomes $\{(x_1 \vee b_1, 1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1), (x_5 \vee b_5, 1), (x_6 \vee b_6, 1), (\neg x_6 \vee b_7, 1)\} \cup \phi_H$. Also, the variables *k*, *CurrBit* and *cost* are initialized to 2, 2 and 2^2 respectively. The following are the iterations *BitBased-WPMaxSAT* executes.

1. The constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 < 2^2)$ is included, *state* = *False*, *CurrBit* = 1, *cost* = $2^2 + 2^1 = 6$.
2. The constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 < 2^2 + 2^1)$, *state* = *True*, $I = \{x_1 = \text{False}, x_2 = \text{False}, x_3 = \text{True}, x_4 = \text{False}, x_5 = \text{True}, x_6 = \text{False}, b_1 = \text{True}, b_2 = \text{True}, b_3 = \text{False}, b_4 = \text{True}, b_5 = \text{False}, b_6 = \text{True}, b_7 = \text{False}\}$, *CurrBit* = -1.

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 [18] (see algorithm 6). The algorithms described in this section are

1. Fu and Malik's algorithm [18]
2. WPM1 [4]
3. Improved WPM1 [5]
4. WPM2 [7]
5. WMSU1-ROR [21]
6. WMSU3 [37]
7. WMSU4 [38]

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 [15].

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 WPMaxSAT instance $\phi = \{(C_1, w_1), \dots, (C_s, w_s)\} \cup \{(C_{s+1}, \infty), \dots, (C_{s+h}, \infty)\}$, 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

$$\phi_k = \{(C_i \vee b_i), \dots, (C_s \vee b_s), C_{s+1}, \dots, C_{s+h}\} \cup \text{CNF} \left(\sum_{i=1}^s w_i b_i \leq k \right)$$

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 WPMaXSAT 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[44]. Their PMaxSAT solvers participated in the first and second MaxSAT Evaluations[10]. Their method (algorithm 6) basis for many WPMaXSAT 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.

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

```

Input:  $\phi = \{(C_1, 1), \dots, (C_s, 1), (C_{s+1}, \infty), \dots, (C_{s+h}, \infty)\}$ 
Output: The cost of the optimal assignment to  $\phi$ 
1 if  $\text{SAT}(\{C_{s+1}, \dots, C_{s+h}\}) = (\text{False}, -)$  then
2   return  $\infty$ 
3  $opt \leftarrow 0$  // The cost of the optimal solution
4  $f \leftarrow 0$  // The number of clauses falsified
5 while True do
6    $(state, \phi_C) \leftarrow \text{SAT}(\{C_i \mid (C_i, w_i) \in \phi\})$ 
7   if  $state = \text{True}$  then
8     return  $opt$ 
9    $f \leftarrow f + 1$ 
10   $B \leftarrow \emptyset$ 
11  foreach  $C_i \in \phi_C$  such that  $w_i \neq \infty$  do
12    let  $b_i$  be a new blocking variable
13     $\phi \leftarrow \phi \setminus \{(C_i, 1)\} \cup \{(C_i \vee b_i, 1)\}$ 
14     $B \leftarrow B \cup \{i\}$ 
15   $\phi \leftarrow \phi \cup \{(C, \infty) \mid C \in \sum_{i \in B} b_i = 1\}$  // Add the cardinality constraint as hard clauses
16   $opt \leftarrow opt + 1$ 

```

Fu&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, 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[4] extended Fu& Malik to WPMMaxSAT. The resulting algorithm is called WPM1 and is described in algorithm 7.

Algorithm 7: WPM1(ϕ) The WPM1 algorithm for WPMMaxSAT.	
Input: A WPMMaxSAT instance $\phi = \{(H_1, \infty), \dots, (H_h, \infty)\} \cup \{(S_1, w_1), \dots, (S_s, w_s)\}$	
Output: The optimal cost of the WPMMaxSAT solution	
1	if $SAT(\{H_i \mid 1 \leq i \leq h\}) = \text{False}$ then
2	return ∞
3	$cost \leftarrow 0$
4	while <i>True</i> do
5	$(state, \phi_C) \leftarrow SAT(\{C_i \mid (C_i, w_i) \in \phi\})$
6	if $state = \text{True}$ then
7	return $cost$
8	$BV \leftarrow \emptyset$
9	$w_{min} \leftarrow \min\{w_i \mid C_i \in \phi_C \text{ and } w_i \neq \infty\}$
	// Compute the minimum weight of all the soft clauses in ϕ_C
10	foreach $C_i \in \phi_C$ do
11	if $w_i \neq \infty$ then
12	Let b_i be a new blocking variable
13	$\phi \leftarrow \phi \setminus \{(C_i, w_i)\} \cup \{(C_i, w_i - w_{min})\} \cup \{(C_i \vee b_i, w_{min})\}$
14	$BV \leftarrow BV \cup \{b_i\}$
15	if $BV = \emptyset$ then
16	return <i>False</i> // ϕ is unsatisfiable
17	else
18	$\phi \leftarrow \phi \cup CNF(\sum_{b \in BV} b = 1)$ // Add the cardinality constraint as hard clauses
19	$cost \leftarrow cost + w_{min}$

Just as in Fu&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 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, $\phi_C = \{C_1, \dots, C_k\}$, appears l times, all the copies get the same set of blocking variables. This is possible because the two formulae $\phi_1 = \phi \setminus \phi_C \cup \{C_1 \vee b_i, \dots, C_i \vee b_i \mid C_i \in \phi_C\} \cup CNF\left(\sum_{i=1}^k b_i = 1\right)$ and $\phi_2 = \phi \setminus \phi_C \cup \{C_i \vee b_i^1, \dots, C_i \vee b_i^l \mid C_i \in \phi_C\} \cup CNF\left(\sum_{i=1}^k b_i^1 = 1\right) \cup \dots \cup CNF\left(\sum_{i=1}^k b_i^l = 1\right)$ 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. This disadvantage is eliminated by WMSU3 (described later).

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 with

$$\sum_{\substack{C_i \text{ is soft} \\ A \text{ satisfies } C_i}} w_i = 2$$

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)[5]. 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 $\{(x_1 \vee b_1^1 \vee \dots \vee b_1^i, 1), (x_2 \vee b_2^1, 1), \dots, (x_2 \vee b_2^i, 1), (x_2, w - i), (\neg x_2, \infty), (b_1^1 + b_2^1 = 1, \infty), \dots, (b_1^i + b_2^i = 1, \infty)\}$. In this case, WPM1 would need w iterations to solve the problem.

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

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

```

1 if SAT( $\{C_i \mid w_i = \infty\}$ ) = (False,  $\_$ ) then
2   return  $\infty$  // cost =  $\infty$  if the hard clauses can not be satisfied
3 cost  $\leftarrow$  0
4  $w_{max} \leftarrow \max\{w_i \mid (C_i, w_i) \in \phi \text{ and } w_i < \infty\}$  // Initialize  $w_{max}$  to the largest
   weight smaller than  $\infty$ 
5 while True do
6   ( $state, \phi_C$ )  $\leftarrow$  SAT( $\{C_i \mid (C_i, w_i) \in \phi \text{ and } w_i \geq w_{max}\}$ )
7   if  $state = \text{True}$  and  $w_{max} = 0$  then
8     return cost
9   else
10    if  $state = \text{True}$  then
11       $w_{max} \leftarrow \max\{w_i \mid (C_i, w_i) \in \phi \text{ and } w_i < w_{max}\}$ 
12    else
13       $BV \leftarrow \emptyset$  // Set of blocking variables of the unsatisfiable core
14       $w_{min} \leftarrow \min\{w_i \mid C_i \in \phi_C \text{ and } w_i \neq \infty\}$  // Minimum weight of soft
        clauses in the unsatisfiable core
15      foreach  $C_i \in \phi_C$  do
16        if  $w_i \neq \infty$  then
17          Let  $b$  be a new variable
18           $\phi \leftarrow \phi \setminus \{(C_i, w_i)\} \cup \{(C_i, w_i - w_{min}), (C_i \vee b, w_{min})\}$ 
19           $BV \leftarrow BV \cup \{b\}$ 
19       $\phi \leftarrow \phi \cup \{(C, \infty) \mid C \in CNF(\sum_{b \in BV} b = 1)\}$  // The cardinality
        constraint is added as hard clauses
20      cost  $\leftarrow$  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[37] discussed important properties of Fu&Malik that were not mentioned in[18]. 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&Malik use the pairwise encoding[19] 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[57, 53, 17, 19].

Another drawback of Fu&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 WPMMaxSAT in 2010, called WPM2[7], 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 on 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 $\{1, \dots, m\}$ such that for every $A \in L$ and $B \in Covers(L)$, if $A \cap B \neq \emptyset$, then $A \subseteq B$.

Algorithm 9: WPM2(ϕ) The WPM2 algorithm for WPMMaxSAT

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

```

1 if SAT( $\{C_i \in \phi \mid w_i = \infty\}$ ) = (False, -) then
2   return  $\infty$ 
3  $\phi^e \leftarrow \{C_1 \vee b_1, \dots, C_m \vee b_m, C_{m+1}, \dots, C_{m+m'}\}$ 
4  $Covers \leftarrow \{\{1\}, \dots, \{m\}\}$ 
5  $AL \leftarrow \emptyset$ 
6  $AM \leftarrow \{w_1 b_1 \leq 0, \dots, w_m b_m \leq 0\}$ 
7 while True do
8    $(state, \phi_C, I) \leftarrow SAT(\phi^e \cup CNF(AL \cup AM))$ 
9   if  $state = \text{True}$  then
10    return  $I$ 
11   Remove the hard clauses from  $\phi_C$ 
12   if  $\phi_C = \emptyset$  then
13    return  $\emptyset$  //  $\phi$  has no solution
14    $A \leftarrow \emptyset$ 
15   foreach  $C_i \vee b_i \in \phi_C$  do
16     $A \leftarrow A \cup \{i\}$ 
17    $RC \leftarrow \{B \in Covers \mid B \cap A \neq \emptyset\}$ 
18    $B \leftarrow \bigcup_{B' \in RC} B'$ 
19    $k \leftarrow NewBound(AL, B)$ 
20    $Covers \leftarrow Covers \setminus RC \cup B$ 
21    $AL \leftarrow AL \cup \{\sum_{i \in B} w_i b_i \geq k\}$ 
22    $AM \leftarrow AM \setminus \{\sum_{i \in B'} w_i b_i \leq k' \mid B' \in RC\} \cup \{\sum_{i \in B} w_i b_i \leq k\}$ 

```

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 .

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 $Core(\sum_{i \in A} w_i b_i \geq k)$ returns the core A , and $Cores(AL)$ returns $\{Core(al) \mid al \in AL\}$.

Definition 4.6 (Disjoint cores). Let $U = \{U_1, \dots, U_k\}$ 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 $(R_i \cap R_j = \emptyset \text{ and } i \neq j)$

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, $AL = \{w_1b_1 \geq 0, \dots, w_mb_m \geq 0\}$ and the corresponding $AM = \{w_1b_1 \leq 0, \dots, w_mb_m \leq 0\}$ which ensures that the solution to $AL \cup AM$ is $b_1 = False, \dots, b_m = False$. 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 $A \subseteq \{1, \dots, m\}$ 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 $\sum_{i \in B} w_ib_i \geq NewBound(AL, B)$, where $NewBound(AL, B)$ correspond to minimize $\sum_{i \in A} w_ib_i$ subject to the set of constraints $\{\sum_{w_ib_i \geq k}\} \cup AL$ where $k = 1 + \sum\{k' \mid \sum_{i \in A'} w_ib_i \leq k' \in AM \text{ and } A' \subseteq A\}$. 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 $\{w_1, \dots, w_n\}$ and k , minimize $\sum_{j=1}^n w_jx_j$ subject to $\sum_{j=1}^n w_jx_j > k$ and $x_j \in \{0, 1\}$. This is equivalent to $NewBound(AL, B)$, where the weights are w_j , $B = \{1, \dots, n\}$ and $AL = \{\sum_{j=1}^n w_jx_j \geq k\}$. In the authors' implementation, $NewBound$ is computed by algorithm 10.

Algorithm 10: $NewBound(AL, B)$

```

1  $k \leftarrow \sum\{k' \mid \sum_{i \in B'} w_ib_i \leq k' \in AM \text{ and } B' \subseteq B\}$ 
2 repeat
3    $k \leftarrow SubsetSum(\{w_i \mid i \in B\}, k)$ 
4 until  $SAT(CNF(AL \cup \{\sum_{i \in B} w_ib_i = k\}))$ 
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 .

Example 4.2. Consider ϕ in example 2.1 with all the weights of the soft clauses set to 1. Before the main loop of algorithm 9, we have $\phi^e = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5 \vee b_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\} \cup \phi_H$, $Covers = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}\}$, $AL = \emptyset$, $AM = \{b_1 \leq 0, b_2 \leq 0, b_3 \leq 0, b_4 \leq 0, b_5 \leq 0, b_6 \leq 0, b_7 \leq 0\}$. The following are the iterations the algorithm executes. The soft clauses in the core ϕ_C are denoted by $Soft(\phi_C)$.

1. $state = False$, $Soft(\phi_C) = \{(x_6 \vee b_6), (\neg x_6 \vee b_7)\}$, $A = \{6, 7\}$, $RC = \{\{6\}, \{7\}\}$, $B = \{6, 7\}$, $k = 1$, $Covers = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6, 7\}\}$, $AL = \{b_6 + b_7 \geq 1\}$, $AM = \{b_1 \leq 0, b_2 \leq 0, b_3 \leq 0, b_4 \leq 0, b_5 \leq 0, b_6 + b_7 \leq 1\}$.
2. $state = False$, $Soft(\phi_C) = \{(x_1), (x_2)\}$, $A = \{1, 2\}$, $RC = \{\{1\}, \{2\}\}$, $B = \{1, 2\}$, $k = 1$, $Covers = \{\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6, 7\}\}$, $AL = \{b_6 + b_7 \geq 1, b_1 + b_2 \geq 1\}$, $AM = \{b_3 \leq 0, b_4 \leq 0, b_5 \leq 0, b_6 + b_7 \leq 1, b_1 + b_2 \leq 1\}$.
3. $state = False$, $Soft(\phi_C) = \{(x_3), (x_4)\}$, $A = \{3, 4\}$, $RC = \{\{3\}, \{4\}\}$, $B = \{3, 4\}$, $k = 1$, $Covers = \{\{1, 2\}, \{3, 4\}, \{5\}, \{6, 7\}\}$, $AL = \{b_6 + b_7 \geq 1, b_1 + b_2 \geq 1, b_3 + b_4 \geq 1\}$, $AM = \{b_1 + b_2 \leq 1, b_5 \leq 0, b_6 + b_7 \leq 1, b_3 + b_4 \leq 1\}$.

4. $state = False$, $Soft(\phi_C) = \{(x_1), (x_2), (x_3), (x_4), (x_5)\}$, $A = \{1, 2, 3, 4, 5\}$, $RC = \{\{1, 2\}, \{3, 4\}, \{5\}\}$, $B = \{1, 2, 3, 4, 5\}$, $k = 3$, $Covers = \{\{6, 7\}, \{1, 2, 3, 4, 5\}\}$, $AL = \{b_6 + b_7 \geq 1, b_1 + b_2 \geq 1, b_3 + b_4 \geq 1, b_1 + b_2 + b_3 + b_4 + b_5 \geq 3\}$, $AM = \{b_1 + b_2 \leq 1, b_1 + b_2 + b_3 + b_4 + b_5 \leq 3\}$.
5. $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = False, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$.

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[6], Ansótegui *et al.* invented three improvements to WPM2. First, they applied the stratification technique[5]. 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 WPM2 instance, the 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 WMSU1-ROR

WMSU1-ROR[21] is a modification of WPM1. It attempts to avoid adding blocking variables by applying MaxSAT resolution to the clauses of the unsatisfiable core. Given an unsatisfiable core ϕ_C , a resolution refutation (a contradiction obtained by performing resolution) is calculated by a specialized tool. As much of this refutation as possible is copied by applying MaxSAT resolution steps to the working formula. If the transformation derived the empty clause, it means that the core is trivial and the sequence of calls to the SAT solver can continue without adding any relaxation variables for this step. Otherwise, the transformed core is relaxed as in WPM1. The classical resolution rule can not be applied in MaxSAT because it does not preserve the equivalence among weighted formulae. The MaxSAT resolution rule used in WMSU1-ROR is called Max-RES and is described in[26]. The following definition extends the resolution rule from SAT to WMaxSAT.

Definition 4.7 (WPM2 resolution). $\{(x \vee A, u), (\neg x \vee B, w)\} \equiv \{(A \vee B, m), (x \vee A, u \odot m), (\neg x \vee B, w \odot m), (x \vee A \vee \neg B, m), (\neg x \vee \neg A \vee B, m)\}$, where A and B are disjunctions and \odot is defined on weights $u, w \in \{0, \dots, \top\}$, such that $u \geq w$, as

$$u \odot w = \begin{cases} u - w & u \neq \top \\ \top & u = \top \end{cases}$$

and $m = \min(u, w)$. The clauses $(x \vee A, u)$ and $(\neg x \vee B, w)$ are called the *clashing clauses*, $(A \vee B, m)$ is called the *resolvent*, $(x \vee A, u \odot m)$ and $(\neg x \vee B, w \odot m)$ are called *posterior clashing clauses*, $(x \vee A \vee \neg B, m)$ and $(\neg x \vee \neg A \vee B, m)$ are the *compensation clauses* (which are added to recover an equivalent MaxSAT formula).

For example, if Max-RES is applied on $\{(x \vee y, 3), (\neg x \vee y \vee z, 4)\}$ with $\top > 4$, we obtain $\{(y \vee y \vee z, 3), (x \vee y, 3 \odot 3), (\neg x \vee y \vee z, 4 \odot 3), (x \vee y \vee \neg(y \vee z), 3), (\neg x \vee \neg y \vee y \vee z, 3)\}$.

The first and fourth clauses can be simplified by observing that $(A \vee C \vee \neg(C \vee B), u) \equiv (A \vee C \vee \neg B, u)$. The second and fifth clauses can be deleted since the former has weight zero and the latter is a tautology. De Morgan's laws can not be applied on MaxSAT instance for not preserving the equivalence among instances[26]. The following rule can be applied instead $(A \vee \neg(l \vee C), w) \equiv \{(A \vee \neg C), (A \vee \neg l \vee C, w)\}$. A resolution proof is an ordered set $R = \{C_i = (C_{i'} \bowtie C_{i''}), C_{i+1} = (C_{i'+1} \bowtie C_{i''+1}), \dots, C_{i+k} = (C_{i'+k} \bowtie C_{i''+k})\}$, where $(C_i, w_i) = (C_{i'}, w_{i'}) \bowtie (C_{i''}, w_{i''})$ is the resolution step i of a resolution proof, (C_i, w_i) is the resolvent and $(C_{i'}, w_{i'})$ and $(C_{i''}, w_{i''})$ are the clashing clauses. The set of compensation clauses will be denoted $[(C_{i'}, w_{i'}) \bowtie (C_{i''}, w_{i''})]$.

The ROR approach is captured in lines 12-22 in algorithm 11. WMSU1-ROR handles WPMaXSAT formulae the same way as[4]. It maintains a working formula ϕ_W and a lower bound LB . The resolution proof R_C is obtained in line 12 and MaxSAT resolution is applied (lines 14-21) for each read-once step. In detail, the weights of the clashing clauses $(C_{i'}, w_{i'})$ and $(C_{i''}, w_{i''})$ are decreased by the minimum weight of the clauses in the unsatisfiable core ϕ_C (lines 15-16). If the clashing clauses are soft, they are deleted from ϕ_C (lines 17-18) and if their resolvent is not \square , it is added to ϕ_C (lines 21-22). On the other hand, if the clashing clauses are hard, they are kept in the core because they could be used in a different resolution step. Lastly, the compensation and clashing clauses are added to ϕ_W (lines 19-20).

Algorithm 11: WMSU1-ROR(ϕ)

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

```
1 if SAT( $\{C_i \mid w_i = \infty\}$ ) = False then
2   return  $\infty$ 
3  $\phi_W \leftarrow \phi$ 
4  $LB \leftarrow 0$ 
5 while True do
6   ( $state, \phi_C$ )  $\leftarrow$  SAT( $\phi_W$ )
7   if  $state = \text{True}$  then
8     return  $LB$ 
9    $\phi_W \leftarrow \phi_W \setminus \phi_C$ 
10   $m \leftarrow \min(\{w \mid (C, w) \in \phi_C \text{ and } w < \top\})$ 
11   $LB \leftarrow LB + m$ 
    // Beginning of read-once resolution
12   $R_C \leftarrow \text{GetProof}(\phi_C)$ 
13  foreach  $(C_i, w_i) = (C_{i'}, w_{i'}) \bowtie (C_{i''}, w_{i''}) \in R_C$  do
14    if ROR( $(C_i, w_i), R_C$ ) then
15       $(C_{i'}, w_{i'}) \leftarrow (C_{i'}, w_{i'} \ominus m)$ 
16       $(C_{i''}, w_{i''}) \leftarrow (C_{i''}, w_{i''} \ominus m)$ 
17      if  $w_{i'} < \top$  and  $w_{i''} < \top$  then
18         $\phi_C \leftarrow \phi_C \setminus \{(C_{i'}, w_{i'}), (C_{i''}, w_{i''})\}$ 
19         $\phi_W \leftarrow \phi_W \cup \{(C_{i'}, w_{i'}), (C_{i''}, w_{i''})\}$ 
20         $\phi_W \leftarrow \phi_W \cup \{[(C_{i'}, w_{i'}) \bowtie (C_{i''}, w_{i''})]\}$ 
21        if  $C_i \neq \square$  then
22           $\phi_C \leftarrow \phi_C \cup \{(C_i, m)\}$ 
    // End of read-once resolution
23   $B \leftarrow \emptyset$ 
24  foreach  $(C_i, w_i) \in \{(C, w) \mid (C, w) \in \phi_C \text{ and } w < \top\}$  do
25    Let  $b$  be a new relaxation variable  $B \leftarrow B \cup \{b\}$   $\phi_C \leftarrow \phi_C \cup \{(C \vee b, m)\}$  if  $w > m$ 
    then
26       $(C, w) \leftarrow (C, w \ominus m)$ 
27    else
28       $\phi_C \leftarrow \phi_C \setminus \{(C, w)\}$ 
29   $\phi_c \leftarrow \phi_C \cup CNF(\sum_{b \in B} b = 1)$ 
30   $\phi_W \leftarrow \phi_W \cup \phi_C$ 
```

$Hard((C_i, w_i), R)$ (algorithm 12) returns *True* if (C_i, w_i) is a hard clause and all its ancestors are hard, otherwise it returns *False*. $Input((C_i, w_i), R)$ (called in line 1) returns *True* if (C_i, w_i) is not a resolvent of any step in R (i.e., an original clause), otherwise it returns *False*. $ancestors((C_i, w_i), R)$ (called in line 5) returns the pair of clauses $(C_{i'}, w_{i'})$ and $(C_{i''}, w_{i''})$ from which (C_i, w_i) was derived as dictated by R .

Algorithm 12: $\text{Hard}((C_i, w_i), R)$ Determines if a clause is hard or not

Input: A proof R
Output: *True* if (C_i, w_i) is hard, or *False* otherwise

```

1 if  $\text{Input}((C_i, w_i), R)$  and  $w_i = \top$  then
2   return True
3 if  $\text{Input}((C_i, w_i), R)$  and  $w_i \neq \top$  then
4   return False
5  $\{(C_{i'}, w_{i'}), (C_{i''}, w_{i''})\} \leftarrow \text{ancestors}((C_i, w_i), R)$ 
6 return  $\text{Hard}((C_{i'}, w_{i'}), R)$  and  $\text{Hard}((C_{i''}, w_{i''}), R)$ 

```

The function ROR (algorithm 13) returns *True* if (C_i, w_i) is hard or if it and all of its soft ancestors have been used at most once in the resolution proof R . If $(C_k, w_k) = (C_{k'}, w_{k'}) \boxtimes (C_{k''}, w_{k''})$, where (C_k, w_k) is the last resolvent in a resolution proof R . The entire proof is read-once if $ROR((C_k, w_k), R)$ returns *True*. In this case (when the last step is ROR), the resolvent of that step is (\square, m) . If this situation occurs, the algorithm does not need to augment clauses with relaxation variables or cardinality constraints, which improves upon the original algorithm.

Algorithm 13: $\text{ROR}((C_i, w_i), R)$ Determines if a clause is hard or not or if its ancestors are used at most once

Input: A proof R
Output: *True* if (C_i, w_i) is hard or if its ancestors are used exactly once, *False* otherwise

```

1 if  $\text{Hard}((C_i, w_i), R)$  then
2   return True
3 if  $\text{Input}((C_i, w_i), R)$  and  $\text{Used}((C_i, w_i), R) = 1$  then
4   return True
5 if  $\text{Used}((C_i, w_i), R) > 1$  then
6   return False
7  $\{(C_{i'}, w_{i'}), (C_{i''}, w_{i''})\} \leftarrow \text{ancestors}((C_i, w_i), R)$ 
8 return  $\text{ROR}((C_{i'}, w_{i'}), R)$  and  $\text{ROR}((C_{i''}, w_{i''}), R)$ 

```

The problem with this approach (applying Max-RES instead of adding blocking variables and cardinality constraints) is that when soft clauses with weights greater than zero are resolved more than once, MaxSAT resolution does not ensure to produce resolvents with weights greater than zero. For this technique to work, the authors restrict the application of resolution to the case where each clause is used at most once, which is referred to as *read-once resolution* (ROR). Unfortunately, ROR can not generate resolution proofs for some unsatisfiable clauses[23].

4.6 WMSU3

WMSU3 is a WPMAXSAT algorithm that adds a single blocking variable per soft clause, thus limiting the number of variables in the formula sent to the SAT solver in each iteration.

Algorithm 14: WMSU3(ϕ) The WMSU3 algorithm for WPMaXSAT.

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

```

1 if SAT( $\{C \mid (C, \infty) \in \phi_H\}$ ) = False then
2   return  $\infty$ 
3  $B \leftarrow \emptyset$  // Set of blocking variables
4  $\phi_W \leftarrow \phi$  // Working formula initialized to  $\phi$ 
5  $LB \leftarrow 0$  // Lower bound initialized to 0
6 while True do
7   ( $state, \phi_C$ )  $\leftarrow$  SAT( $\{C \mid (C, w) \in \phi_W\} \cup CNF(\sum_{b_i \in B} w_i b_i \leq LB)$ )
8   if  $state = True$  then
9     return  $LB$ 
10  foreach  $(C_i, w_i) \in \phi_C \cap \phi_S$  do
11    if  $w \neq \infty$  then
12       $B \leftarrow B \cup \{b_i\}$ 
13       $\phi_W \leftarrow \phi_W \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
14   $LB \leftarrow UpdateBound(\{w_i \mid b_i \in B\}, LB)$ 

```

Algorithm 14 begins by initializing the set of blocking variables that will be augmented later to \emptyset (line 3), the working formula to ϕ (line 4) and the lower bound to zero (line 5). MSU3 then loops over unsatisfiable working formulae ϕ_W (while loop in lines 6-13) until it finds a satisfiable one in line 8. At each iteration, when an unsatisfiable core is returned by the SAT solver, the algorithm adds one blocking variable to each soft clause that has not been augmented with a blocking variable yet (line 13), unlike WPMaXSAT algorithms discussed previously such as WPM1 (algorithm 7). Indeed, at most one blocking variable is added to each clause because if at iteration i C_i was blocked by b_i , then at iteration $i+1$ the clause $C_i \vee b_i$ will not be in $\phi_C \cap \phi_S$. The function *UpdateBound* in line 14 updates the lower bound LB , either by simply incrementing it or by the subset sum problem as in[7]. The following example illustrates how the algorithm works.

Example 4.3. Let $\phi = \{(x_1, 1), (x_2, 3), (x_3, 1)\} \cup \{(\neg x_1 \vee \neg x_2, \infty), (\neg x_2 \vee \neg x_3, \infty)\}$.

1. $state = False$, $\phi_C = \{(x_1), (x_2), (\neg x_1 \vee \neg x_2)\}$, $\phi_C \cap \phi_S = \{(x_1), (x_2)\}$, $\phi_W = \{(x_1 \vee b_1, 1), (x_2 \vee b_2, 2), (x_3, 1), (\neg x_1 \vee \neg x_2, \infty), (\neg x_2 \vee \neg x_3, \infty)\}$, $LB = 1$.
2. The constraint $CNF(b_1 + 3b_2 \leq 1)$ is included and satisfying it implies that b_2 must be falsified, and thus $CNF(b_1 + 3b_2 \leq 1)$ is replaced by $(\neg b_2)$. $state = False$, $\phi_C = \{(x_2 \vee b_2), (x_3), (\neg x_2 \vee \neg x_3), (\neg b_2)\}$, $\phi_C \cap \phi_S = \{(x_3)\}$, $\phi_W = \{(x_1 \vee b_1, 1), (x_2 \vee b_2, 2), (x_3 \vee b_3, 1), (\neg x_1 \vee \neg x_2, \infty), (\neg x_2 \vee \neg x_3, \infty)\}$, $LB = 2$. As in the previous iteration, satisfying the constraint $b_1 + 3b_2 + b_3 \leq 2$ implies b_2 must be falsified.
3. The constraint $CNF(b_1 + 3b_2 + b_3 \leq 2)$ is included, $state = True$ and the assignment $I = \{x_1 = False, x_2 = True, x_3 = False, b_1 = True, b_2 = False, b_3 = True\}$ indeed satisfies ϕ_W of the last iteration. By ignoring the values of the blocking variables, I is indeed an optimal assignment for ϕ . It falsifies the soft clauses $(x_1, 1)$ and $(x_3, 1)$ and satisfies $(x_2, 3)$.

4.7 WMSU4

Like WMSU3, WMSU4[38] (algorithm 15) 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 15: WMSU4(ϕ) The WMSU4 algorithm for WPMaXSAT.

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

```

1 if SAT( $\{C \mid (C, \infty) \in \phi_H\}$ ) = False then
2   return  $\infty$ 
3  $B \leftarrow \emptyset$  // Set of blocking variables
4  $\phi_W \leftarrow \phi$  // Working formula initialized to  $\phi$ 
5  $LB \leftarrow -1$  // Lower bound initialized to 0
6  $UB \leftarrow 1 + \sum_{i=1}^{|\phi_S|} w_i$  // Upper bound initialized to the sum of the weights
   of the soft clauses plus one
7 while  $UB > LB + 1$  do
8    $(state, \phi_C, I) \leftarrow SAT(\{C \mid (C, w) \in \phi_W\} \cup CNF(\sum_{b_i \in B} w_i b_i \leq UB - 1))$ 
9   if  $state = \text{True}$  then
10     $UB \leftarrow \sum_{b_i \in B} w_i (1 - I(C_i \setminus b_i))$  // Update  $UB$  to the sum of the
    weights of the falsified clauses without the blocking variables
11  else
12    foreach  $(C_i, w_i) \in \phi_C \cap \phi_S$  do
13      if  $w \neq \infty$  then
14         $B' \leftarrow B' \cup \{b_i\}$ 
15         $\phi_W \leftarrow \phi_W \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
16    if  $B' = \emptyset$  then
17       $LB \leftarrow UB - 1$ 
18    else
19       $B \leftarrow B \cup B'$ 
20       $LB \leftarrow UpdateBound(\{w_i \mid b_i \in B\}, LB)$ 
21 return  $UB$ 

```

Example 4.4. Let $\phi = \phi_S \cup \phi_H$, where $\phi_S = \{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 1)\}$ and $\phi_H = \{(\neg x_1 \vee \neg x_2, \infty), (\neg x_1 \vee \neg x_3, \infty), (\neg x_1 \vee \neg x_4, \infty), (\neg x_2 \vee \neg x_3 \vee \neg x_4, \infty)\}$. Before the first iteration of the while loop, we have $LB = -1$, $UB = 1 + (1 + 1 + 1 + 1) = 5$

and $\phi_W = \phi$.

1. *state = False, $\phi_C \cap \phi_S = \{(x_2), (x_3), (x_4)\}$, $LB = 0$, $\phi_W = \{(x_1, 1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1)\} \cup \phi_H$.*
2. *The constraint $CNF(b_2 + b_3 + b_4 \leq 5 - 1)$ is included, $state = True$, $I = \{x_1 = True, x_2 = False, x_3 = False, x_4 = False, b_2 = True, b_3 = True, b_4 = True\}$, $UB = 3$.*
3. *The constraint $CNF(b_2 + b_3 + b_4 \leq 3 - 1)$ is included, $state = False$, $\phi_C \cap \phi_S = \{(x_1, 1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1)\}$, $LB = 1$, $\phi_W = \{(x_1 \vee b_1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1)\} \cup \phi_H$.*
4. *The constraint $CNF(b_1 + b_2 + b_3 + b_4 \leq 2)$ is included, $state = SAT$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = True, b_1 = True, b_2 = True, b_3 = False, b_4 = False\}$, $UB = 2$. The cost of the optimal assignment is indeed 2 (since $(x_1, 1)$ and $(x_2, 1)$ are falsified) by I .*

5 Core-guided Binary Search Algorithms

Core-guided binary search algorithms are similar to binary search algorithms described in the first section, 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 [22] (see algorithm 16).

Algorithm 16: CoreGuided-BS(ϕ) Core-guided binary search algorithm for solving WPMaXSAT.

Input: A WPMaXSAT instance $\phi = \phi_S \cup \phi_H$
Output: The cost of the optimal WPMaXSAT 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  $\phi_W \leftarrow \phi$ 
5  $LB \leftarrow -1$ 
6  $UB \leftarrow 1 + \sum_{i=1}^{|\phi_S|} w_i$ 
7  $B \leftarrow \emptyset$ 
8 while  $LB + 1 < UB$  do
9    $mid \leftarrow \lfloor \frac{LB+UB}{2} \rfloor$ 
10   $(state, \phi_C, I) \leftarrow SAT(\{C \mid (C, w) \in \phi_W\} \cup CNF(\sum_{b_i \in B} w_i b_i \leq mid))$ 
11  if  $state = True$  then
12     $UB \leftarrow \sum_{i=1}^{|\phi_S|} w_i (1 - I(C_i \setminus \{b_i\}))$ 
13     $lastI \leftarrow I$ 
14  else
15    if  $\phi_C \cap \phi_S = \emptyset$  then
16       $LB \leftarrow UpdateBound(\{w_i \mid b_i \in B\}, mid) - 1$ 
17    else
18      foreach  $(C_i, w_i) \in \phi_C \cap \phi_S$  do
19        let  $b_i$  be a new blocking variable
20         $B \leftarrow B \cup \{b_i\}$ 
21         $\phi_W \leftarrow \phi_W \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
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), 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 soft clauses belonging to the core are relaxed (lines 17-19). The main loop continues as long as $LB + 1 < UB$.

Example 5.1. Consider ϕ in example 2.1 with all the weights of the soft clauses set to 1. At the beginning of the algorithm $LB = -1$, $UB = 8$, $B = \emptyset$ and ϕ_H is satisfiable. The following are the iterations the algorithm executes.

1. $mid = \lfloor \frac{-1+8}{2} \rfloor = 3$. Since $B = \emptyset$, no constraint is included. $state = False$, $\phi_C \cap \phi_S = \{(x_6), (\neg x_6)\}$, $B = \{b_6, b_7\}$. $\phi = \{(x_1, 1), (x_2, 1), (x_3, 1),$

- $(x_4, 1), (x_5, 1), (x_6 \vee b_6, 1), (\neg x_6 \vee b_7, 1)\} \cup \phi_H$.
2. $mid = 3$, the constraint $CNF(b_6 + b_7 \leq 3)$ is included. $state = False$, $\phi_C \cap \phi_S = \{(x_1), (x_2)\}$, $B = \{b_1, b_2, b_6, b_7\}$, $\phi = \{(x_1 \vee b_1, 1), (x_2 \vee b_2, 1), (x_3, 1), (x_4, 1), (x_5, 1), (x_6 \vee b_6, 1), (\neg x_6 \vee b_7, 1)\} \cup \phi_H$.
 3. $mid = 3$, the constraint $CNF(b_1 + b_2 + b_6 + b_7 \leq 3)$ is included. $state = False$, $\phi_C \cap \phi_S = \{(x_3), (x_4)\}$, $B = \{b_1, b_2, b_3, b_4, b_6, b_7\}$, $\phi = \{(x_1 \vee b_1, 1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1), (x_5, 1), (x_6 \vee b_6, 1), (\neg x_6 \vee b_7, 1)\} \cup \phi_H$.
 4. $mid = 3$, the constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_6 + b_7 \leq 3)$ is included. $state = False$, $\phi_C \cap \phi_S = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_6 \vee b_6), (\neg x_6 \vee b_7), (x_5)\}$, $B = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$, $\phi = \{(x_1 \vee b_1, 1), (x_2 \vee b_2, 1), (x_3 \vee b_3, 1), (x_4 \vee b_4, 1), (x_5 \vee b_5, 1), (x_6 \vee b_6, 1), (\neg x_6 \vee b_7, 1)\} \cup \phi_H$.
 5. $mid = 3$, $CNF(b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7 \leq 3)$ is included. $state = False$, $\phi_C \cap \phi_S = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5 \vee b_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\}$, $LB = 3$.
 6. $mid = 5$, the constraint $CNF(b_1 + b_2 + b_3 + b_4 + b_6 + b_7 \leq 5)$ is included. $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = False, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$, $UB = 4$. The values of the $x_i, (1 \leq i \leq 6)$ variables in I indeed constitute an optimal assignment.

The core-guided binary search approach was improved by Heras[22] *et al.* with disjoint cores (see definition 4.6).

Algorithm 17: DisjointCoreGuided-BS(ϕ) Core-guided binary search extended with disjoint cores for solving WPMMaxSAT.

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

```

1 if SAT( $\{C \mid (C, \infty) \in \phi_H\}$ ) = False then
2   return  $\emptyset$ 
3  $\phi_W \leftarrow \phi$ 
4  $\mathcal{C} \leftarrow \emptyset$ 
5 repeat
6   foreach  $C_i \in \mathcal{C}$  do
7     if  $LB_i + 1 = UB_i$  then
8        $mid_i \leftarrow UB_i$ 
9     else
10       $mid_i \leftarrow \lfloor \frac{LB_i + UB_i}{2} \rfloor$ 
11   $(state, \phi_C, I) \leftarrow SAT(\{C \mid (C, w) \in \phi_W\} \cup \bigcup_{C_i \in \mathcal{C}} CNF(\sum_{b_i \in B} w_i b_i \leq mid_i))$ 
12  if state = True then
13     $lastI \leftarrow I$ 
14    foreach  $C_i \in \mathcal{C}$  do
15       $UB_i \leftarrow \sum_{b_r \in B} w_r (1 - I(C_r \setminus \{b_r\}))$ 
16  else
17     $subC \leftarrow IntersectingCores(\phi_C, \mathcal{C})$ 
18    if  $\phi_C \cap \phi_S = \emptyset$  and  $|subC| = 1$  then
19       $LB \leftarrow mid$  //  $subC = \{(B, LB, mid, UB)\}$ 
20    else
21      foreach  $(C_i, w_i) \in \phi_C \cap \phi_S$  do
22        let  $b_i$  be a new blocking variable
23         $B \leftarrow B \cup \{b_i\}$ 
24         $\phi_W \leftarrow \phi_W \setminus \{(C_i, w_i)\} \cup \{(C_i \vee b_i, w_i)\}$ 
25       $LB \leftarrow 0$ 
26       $UB \leftarrow 1 + \sum_{b_i \in B} w_i$ 
27      foreach  $(B_i, LB_i, mid_i, UB_i) \in subC$  do
28         $B \leftarrow B \cup B_i$ 
29         $LB \leftarrow LB + LB_i$ 
30         $UB \leftarrow UB + UB_i$ 
31       $\mathcal{C} \leftarrow \mathcal{C} \setminus subC \cup \{(B, LB, 0, UB)\}$ 
32 until  $\forall C_i \in \mathcal{C} UB_i \leq LB_i + 1$ 
33 return  $lastI$ 

```

Core-guided binary search methods with disjoint unsatisfiable cores maintains 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 *IntersectionCores*, every core that intersects the current core is identified and its corresponding entry is added to *subC* (line 17). If the core does not contain soft clauses that need to be relaxed and $|subC| = 1$ (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 *subC* (lines 25-31).

Example 5.2. Consider ϕ in example 2.1 with all the weights of the soft clauses set to 1. At the beginning of algorithm 17, we have $\phi_W = \phi$ and $\mathcal{C} = \emptyset$. The following are the iterations the algorithm executes.

1. No constraints to include. $state = False$, $\phi_C \cap \phi_S = \{(x_6), (\neg x_6)\}$, $subC = \emptyset$, $B = \{b_6, b_7\}$, $\phi_W = \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\} \cup \phi_H$, $LB = 0$, $UB = 3$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 3)\}$.
2. The constraint $CNF(b_6 + b_7 \leq 1)$ is included. $state = False$, $\phi_C \cap \phi_S = \{(x_1), (x_2)\}$, $subC = \emptyset$, $B = \{b_1, b_2\}$. $\phi_W = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3), (x_4), (x_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\} \cup \phi_H$, $LB = 0$, $UB = 3$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 3), (\{b_1, b_2\}, 0, 0, 3)\}$.
3. The constraints $\{CNF(b_6 + b_7 \leq 1), CNF(b_1 + b_2 \leq 1)\}$ are included. $state = False$, $\phi_C \cap \phi_S = \{(x_3), (x_4)\}$, $subC = \emptyset$, $B = \{b_3, b_4\}$, $\phi_W = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\} \cup \phi_H$, $LB = 0$, $UB = 3$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 3), (\{b_1, b_2\}, 0, 0, 3), (\{b_3, b_4\}, 0, 0, 3)\}$.
4. The constraints $\{CNF(b_6 + b_7 \leq 1), CNF(b_1 + b_2 \leq 1), CNF(b_3 + b_4 \leq 1)\}$ are included. $state = False$, $\phi_C \cap \phi_S = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5)\}$, $subC = \{(\{b_1, b_2\}, 0, 0, 3), (\{b_3, b_4\}, 0, 0, 3)\}$, $B = \{b_1, b_2, b_3, b_4, b_5\}$, $\phi_W = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5 \vee b_5), (x_6 \vee b_6), (\neg x_6 \vee b_7)\} \cup \phi_H$, $LB = 0$, $UB = 8$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 3), (\{b_1, b_2, b_3, b_4, b_5\}, 0, 0, 8)\}$.
5. The constraints $CNF(b_6 + b_7 \leq 1), CNF(b_1 + b_2 + b_3 + b_4 + b_5 \leq 4)$ are included. $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = False, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 1), (\{b_1, b_2, b_3, b_4, b_5\}, 0, 0, 2)\}$.
6. The constraints $CNF(b_6 + b_7 \leq 1), CNF(b_1 + b_2 + b_3 + b_4 + b_5 \leq 1)$ are included. $state = False$, $\phi_C \cap \phi_S = \{(x_1 \vee b_1), (x_2 \vee b_2), (x_3 \vee b_3), (x_4 \vee b_4), (x_5 \vee b_5)\}$, $subC = \{(\{b_1, b_2, b_3, b_4, b_5\}, 0, 0, 2)\}$, $\mathcal{C} = \{(\{b_6, b_7\}, 0, 0, 1), (\{b_1, b_2, b_3, b_4, b_5\}, 1, 0, 2)\}$.
7. $state = True$, $I = \{x_1 = False, x_2 = False, x_3 = True, x_4 = False, x_5 = True, x_6 = False, b_1 = True, b_2 = True, b_3 = False, b_4 = True, b_5 = False, b_6 = True, b_7 = False\}$.

SAT-based WPMaXSAT 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. In other words, different solution approaches work well on different families of instances[40]. 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[59] 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[9] (Instance-Specific Algorithm Configuration) is one of the most successful WP-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 Translating Pseudo-Boolean Constraints into CNF

This section discusses translating pseudo-Boolean (PB) constraints into CNF. The procedure is needed in almost every SAT-based WPMaXSAT algorithm and its efficiency surely affects the overall performance of the solver.

7.1 Introduction

A *PB constraint* is a linear constraint over Boolean variables. PB constraints are intensively used in expressing NP-hard problems. While there are dedicated solvers (such as Sat4j) for solving PB constraints, there are good reasons to be interested in transforming the constraints into SAT (CNF formulae), and a number of methods for doing this have been reported[53, 12, 36, 2, 55, 33, 1, 13].

Definition 7.1 (PB constraint). A PB constraint is an inequality (equality) on a linear combination of Boolean literals l_i

$$\sum_{i=1}^n a_i l_i \{<, \geq, =, \leq, >\} K$$

where a_1, \dots, a_n and K (called the bound) are constant integers and l_1, \dots, l_n are literals.

There are at least two clear benefits of solving PB constraints by encoding them into CNF. First, high-performance SAT solvers are being enhanced continuously, and since they take a standard input format there is always a selection of good solvers to make use of. Second, solving problems involving Boolean combinations of constraints is straightforward. This approach is particularly attractive for problems which are naturally represented by a relatively small number of PB constraints (like the Knapsack problem) together with a large number of purely Boolean constraints.

7.2 Encoding method

We present the method of Bailleux, Boufkhad and Roussel[13]. In their paper, they consider (without loss of generality) PB constraints of the form $\sum_{i=1}^n a_i l_i \leq K$, where $a_1 \leq a_2 \leq \dots \leq a_n$. This type of constraint is denoted by the triple $\langle A_n, L_n, K \rangle$, where $A_n = (a_1, \dots, a_n)$ and $L_n = (l_1, \dots, l_n)$. For some bound b , the triple $\langle A_i, L_i, b \rangle$, for $1 \leq i \leq n$, represents the PB constraint $a_1 l_1 + a_2 l_2 + \dots + a_i l_i \leq b$. When the tuples A_n and L_n are fixed, a triple $\langle A_i, L_i, b \rangle$ representing a PB constraint is defined with no ambiguity by the integer i and the bound b .

For each $\langle A_i, L_i, b \rangle$, a new variable $D_{i,b}$ is introduced. This new variable represents the satisfaction of the constraint $\langle A_i, L_i, b \rangle$, i.e., $D_{i,b} = \text{True}$ if and only if $\langle A_i, L_i, b \rangle$ is satisfied. The variable $D_{n,K}$ represents $\langle A_n, L_n, K \rangle$ and the correctness of the encoding is conditioned by the fact that an assignment satisfies $\langle A_n, L_n, K \rangle$ if and only if it satisfies the encoded CNF formula and fixes $D_{n,K}$ to True .

The variables $D_{i,b}$ such that $b \leq 0$ or $b \geq \sum_{j=1}^i a_j$ are called *terminal variables*.

The encoding starts with a set of variables containing the original variables PB constraint and the variable $D_{n,K}$. The variables l_i are marked. At each step, an unmarked variable $D_{i,b}$ is considered. If $D_{i,b}$ is not terminal the two variables $D_{i-1,b}$ and $D_{i-1,b-a_i}$ are added to the set of variables if they are not already in it and the following four clauses are added

$$(\neg D_{i-1,b-a_i} \vee D_{i,b}), (D_{i,b} \vee D_{i-1,b}), (D_{i,b} \vee l_i \vee D_{i-1,b-a_i}), (D_{i-1,b} \vee l_i \vee D_{i,b})$$

Next, $D_{i,b}$ is marked so it won't be considered again.

In case that $D_{i,b}$ is a terminal variable, then by definition either $b \leq 0$ or $b \geq \sum_{j=1}^i a_j$ and $D_{i,b}$ is fixed as follows

$$D_{i,b} = \begin{cases} \text{False} & \text{if } b < 0. \text{ The clause } \neg D_{i,b} \text{ is added to the formula.} \\ \text{True} & \text{if } \sum_{j=1}^i a_j \leq b. \text{ The clause } D_{i,b} \text{ is added to the formula.} \end{cases}$$

When $b = \text{False}$, every variable in the constraint must be set to False . To achieve this, for every $1 \leq j \leq i$, the clauses $(D_{i,0} \vee l_j)$ are added together with the clause $(l_1 \vee l_2 \vee \dots \vee D_{i,0})$. The procedure stops when there are no more unmarked variables.

Example 7.1. This example illustrates the encoding of the PB constraint $2x_1 + 3x_2 + 4x_3 \leq 6$. The formula $\phi = \{(\neg D_{2,2} \vee D_{3,6}), (\neg D_{3,6} \vee \neg x_3 \vee D_{2,2}), (\neg D_{2,6} \vee x_3 \vee D_{3,6}), (D_{2,6} \vee \neg D_{1,-1} \vee D_{2,2}), (\neg D_{2,2} \vee D_{1,2}), (\neg D_{2,2} \vee \neg x_2 \vee D_{1,-1}), (\neg D_{1,2} \vee x_2 \vee D_{2,2}), (D_{1,2}), (\neg D_{1,-1})\}$. Thus, $D_{3,6} = \text{True}$ only if at least one of x_2 or x_3 is False .

The correctness and the complexity of the encoding are discussed in the same paper[13].

7.3 Complexity of the encoding

The complexity of the encoding is measured in terms of the number of variables. The number of clauses produced is related by a constant factor to the number of variables. There are cases where the previous procedure produces a polynomial and others that produce an exponential number of variables.

7.3.1 Polynomial cases

The encoding seems to generate an exponential number of variables: at each step a non-terminal variable creates two variables that will in turn create two other variables each and so on. However, this is not true for terminal variables and for variables that have already been considered by the procedure. When a terminal variable is met, it is said to be a *cut* in the procedure and when a variable already in the set of variables is met, it is said to have *merged* in the procedure. By the cuts and merges, the size of encodings can be polynomial in some cases. There are two restrictions on the PB constraint for it to have a polynomial-size encoding:

1. The integers a_i 's are bounded by a polynomial in n , $P(n)$. In this case, the potential number of $D_{i,b}$ variables for some i is 2^{n-i} but because of the merges, this number reduces to a polynomial since the variables $D_{i,b}$ for some i are such that $m \leq b \leq M$ where m is at least equal to $K - \sum_{j=0}^i a_{n-j}$ and $M \leq K$, b can take at most $M-m$ different values and then it can take at most $\sum_{j=0}^i a_{n-j}$ different values, which is bounded by $(n-i)P(n)$. Since there are n different possible values for i , the total number of variables is bounded by a polynomial in n . Figure 1 shows an example of this case.
2. The weights are $a_i = \alpha_i$ where $\alpha \geq 2$. In this case, for every non terminal variable $D_{i,b}$ considered in the procedure, at least one of the variables $D_{i-1,b}$ or $D_{i-1,b-\alpha^i}$ is a terminal variable. This is true because $\sum_{j=0}^{i-1} \alpha^j < \alpha^i$. Either $b \geq \alpha^i$ and then $\sum_{j=0}^{i-1} \alpha^j < b$ and then $D_{i-1,b}$ is a terminal variable or $b < \alpha^i$ and in this case $D_{i-1,b-\alpha^i}$ is a terminal variable. Thus, there is a cut each time a variable is considered in the procedure. Figure 2 shows an example for this case.

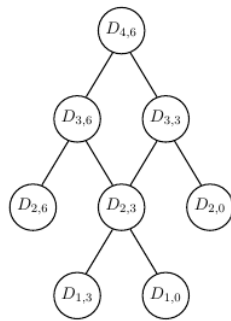


Figure 1: Variables introduced to encode $3x_1 + 3x_2 + 3x_3 + 3x_4 \leq 6$.

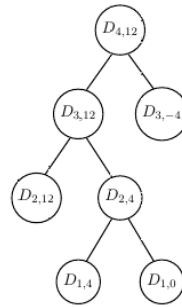


Figure 2: Variables introduced to encode $2x_1 + 4x_2 + 8x_3 + 16x_4 \leq 12$.

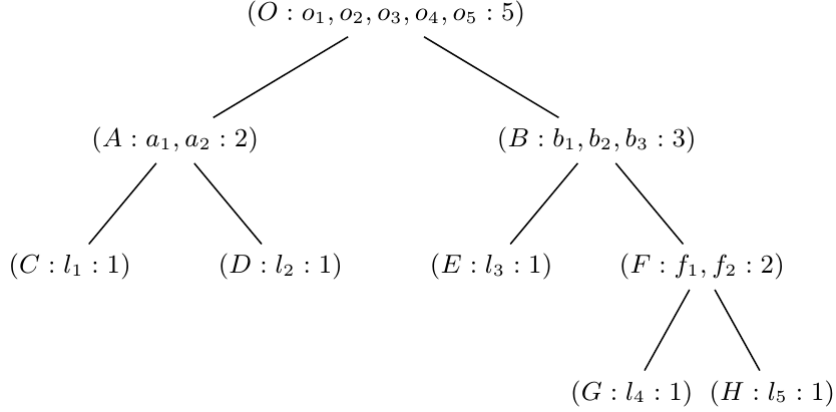


Figure 3: Totalizer encoding for the constraint $l_1 + \dots + l_5 \leq k$

7.3.2 Exponential cases

There are possible sequences of a_i 's that will give a tree with branches of length $\Omega(n)$ and with no possible merge of nodes (which implies a tree of size $\Omega(2n)$). The idea here is simply to combine a constant sequence with a geometric sequence. Let n be the length of the PB constraint Q and let $a_i = \alpha + b^i$ such that $\alpha = b^{n+2}$. The key point is that the geometric term must be negligible compared to the constant term, that is $\sum_{i=0}^n b^i < \alpha$. For simplicity, we will choose $b = 2$. Note that in this case, $a_i = 2^{n+2} + 2^i$ which is not bounded by a polynomial in n . Fix $K = \alpha \times \frac{n}{2} = n \times 2^{n+1}$.

A terminal node is reached when we get a term $D_{i,k}$ such that $k \leq 0$ or $k \geq \sum_{j=1}^i a_j$. Because the constant term is predominant, the first condition cannot be met before $i = \frac{K}{\alpha} = \frac{n}{2}$. The earliest case where the second condition can be satisfied is when k remains equal to K . We have $\sum_{j=1}^i a_j = \sum_{j=1}^i \alpha + b^j = \alpha \times i + \sum_{j=1}^i b^j \geq \alpha \times i$. Therefore, the earliest case where the second condition can be met is when $\alpha \times \frac{n}{2} = \alpha \times i$ which means $i = \frac{n}{2}$. We can conclude that each branch is at least of length $\frac{n}{2}$.

In addition, in the encoding, each node of the tree holds the term $D_{i,k}$ which corresponds to $\sum_{j=1}^i a_j x_j \leq K - \sum_{j \in S} a_j$, where $S \subset [i+1..n]$. One key point is that in the binary representation of $K - \sum_{j \in S} a_j$, the n least significant bits directly correspond to the indices in S . Therefore, these n least significant bits of the right term are necessarily different from one node to another. For this reason, no node can be merged. Because of this and since branches are of length at least equal to $\frac{n}{2}$, the size of the tree is at least $2^{\frac{n}{2}}$ and the encoding of this particular constraint is of exponential size.

7.4 Other encoding techniques

Incremental approaches[42, 39, 47] allow the constraint solver to retain knowledge from previous iterations that may be used in the upcoming iterations. The goal is to retain the inner state of the constraint solver as well as learned clauses that were discovered during the solving process of previous iterations. At each iteration, most MaxSAT algorithms create a new instance of the constraint solver and rebuild the formula losing most if not all the knowledge that could be derived from previous iterations.

8 Experimental Investigation

We conducted an experimental investigation in order to compare the performance of different WPMaXSAT solvers to branch and bound solvers on a number of benchmarks instances.

Experimental evaluations of MaxSAT solvers has gained great interest among SAT and MaxSAT researchers. This is due to the fact that solvers are becoming more and more efficient and adequate to handle WPMaXSAT instances coming from real-life applications. Thus, carrying out such an investigation and comparing the efficiency of different solvers is critical to knowing which solving technique is suitable for which category of inputs. In fact, an annual event called the [MaxSAT Evaluations](#) is scheduled just for this purpose. The [first MaxSAT Evaluation](#) was held in 2006. The objective of the MaxSAT Evaluation is comparing the performance of state of the art (weighted) (partial) MaxSAT solvers on a number of benchmarks and declaring a winner for each benchmark category.

The solvers that we investigate participated in the MaxSAT Evaluations of [2013](#) and [2014](#). A number of the solvers are available online while some of them were not and we had to contact the authors to get a copy. The benchmarks we used participated in the 2013 MaxSAT Evaluation and are WPMaXSAT instances of three categories: random, crafted and industrial.

The solvers were run on a machine with an Intel® Core™ i5 CPU clocked at 2.4GHz, with 5.7GB of RAM running elementary OS Linux. The timeout is set to 1000 seconds and running the solvers on the benchmarks took roughly three months. We picked elementaryOS because it does not consume too many resources to run and thus giving enough room for the solvers to run. In addition, elementaryOS is compatible with popular Ubuntu distribution which makes it compatible with its repositories and packages.

8.1 Solvers descriptions

The solvers we experimented with are:

1. **WMiFuMax** is an unsatisfiability-based WPMaXSAT solver based on the technique of Fu and Malik[18] and on the algorithm by Manquinho, Marques-Silva, and Planes[32], which works by identifying unsatisfiable sub-formulae. MiFuMax placed third in the WPMaXSAT industrial category of the 2013 MaxSAT evaluation. The solver (and the source code) is available online under the GNU General Public License. The SAT solver used is called MiniSAT[54]. Author: Mikoláš Janota.
2. **QWMaxSAT** is a weighted version of QMaxSAT developed by Koshimura, Zhang, Fujita and Hasegawa[25] and is available freely online. This solver is a satisfiability-based solver built on top of version 2.0 of the SAT solver MiniSAT[16]. The authors of QMaxSAT modified only the top-level part of MiniSat to manipulate cardinality constraints, and the other parts remain unchanged. Despite originally being a PMaxSAT solver, the authors developed a version of the solver for WPMaXSAT in 2014. Authors: Miyuki Koshimura, Miyuki Koshimura, Hiroshi Fujita and Ryuzo Hasegawa.
3. **Sat4j**[27] is a satisfiability-based WPMaXSAT solver developed by Le Berre and Parrain. The solver works by translating WPMaXSAT instances into pseudo-Boolean optimization ones. The idea is to add a blocking variable per weighted soft clause that represents that such clause has been violated, and to translate the maximization

problem on those weighted soft clauses into a minimization problem on a linear function over those variables. Given a WPMaXSAT instance $\phi = \{(C_1, w_1), \dots, (C_n, w_n)\} \cup \phi_H$, Sat4j translates ϕ into $\phi' = \{(C_1 \vee b_1), \dots, (C_n \vee b_n)\}$ plus an objective function $\min : \sum_{i=1}^n w_i b_i$. Sat4j avoids adding blocking variables to both hard and unit clauses. the Sat4j framework includes the pseudo-Boolean solver Sat4j-PB-Res which is used to solve the encoded WPMaXSAT problem. Authors: Daniel Le Berre and Emmanuel Lonca.

4. **MSUnCore**[35] is an unsatisfiability-based WPMaXSAT solver built on top the SAT solver PicoSAT[14]. This solver implements a number of algorithms capable of solving MaxSAT, PMaxSAT and W(P)MaxSAT. MSUnCore uses PicoSAT for iterative identification of unsatisfiable cores with larger weights. Although ideally a minimal core would be preferred, any unsatisfiable core can be considered. Clauses in identified core 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[49, 48], the ladder encoding[20], sequential counters[53], sorting networks[17], and binary decision diagrams (BDDs)[17]. Authors: António Morgado, Joao Marques-Silva, and Federico Heras.
5. **Maxsatz2013f** is a very successful 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[28], which incorporates the technique developed for the famous SAT solver, Satz[29]. 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 $\{\square, (\neg x \vee y \vee z), (x \vee \neg y \vee \neg z)\}$. Also, 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. The variable selection heuristics takes into account the number of positive and negative occurrences in binary and ternary clauses. Maxsatz2013f is available freely online. Authors: Chu Min Li, Yanli LIU, Felip Manyà, Zhu Zhu and Kun He.
6. **WMaxSatz-2009** and **WMaxSatz+**[31, 30] are branch and bound solvers that use transformation rules[28] 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.
7. **ISAC+**[9] (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.

Summary		
Technique	Solver name	Sub-technique
Satisfiability-based	WMiFuMax	SAT-based
	QWMaxSAT	SAT-based
	Sat4j	SAT-based
	MSUnCore	UNSAT-based
Branch and bound	Maxsatz2013f	
	WMaxSatz-2009	
	WMaxSatz+	
Portfolio	ISAC+	

8.2 Benchmarks descriptions

The benchmarks we used are the [WPMaXSAT instances](#) of the 2013 MaxSAT Evaluation and are divided into three categories:

1. Random: This category consists of WPMaX-2-SAT and WPMaX-3-SAT instances generated uniformly at random. The WPMaX-2-SAT instances are divided into formulae with low (lo), medium (me) and high (hi) numbers of variables and clauses. The WPMaX-3-SAT instances contain three literals per clause and have a high number of variables and clauses.
2. Crafted: These instances are specifically designed to give a hard time to the solver. There is an award for the smallest instance that can not be solved by any solver.
3. Industrial: Consists of instances that come from various applications of practical interest, such as model checking, planning, encryption, bio-informatics, etc. encoded into MaxSAT. This category is intended to provide a snapshot of the current strength of solvers as engines for SAT-based applications.

In the MaxSAT Evaluations, a first, second and third place winners are declared for each of the three categories.

8.3 Results

In this section, the results we obtained are presented and discussed. 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.

8.3.1 Random category

The three sets of instances in the random category are:

Name	Abbreviation	# of instances
wpmx2sat-lo	lo	30
wpmx2sat-me	me	30
wpmx2sat-hi	hi	30
wpmx3sat-hi	3hi	30

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 1: Number 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%

Table 2: Percentages of instances solved in the random category.

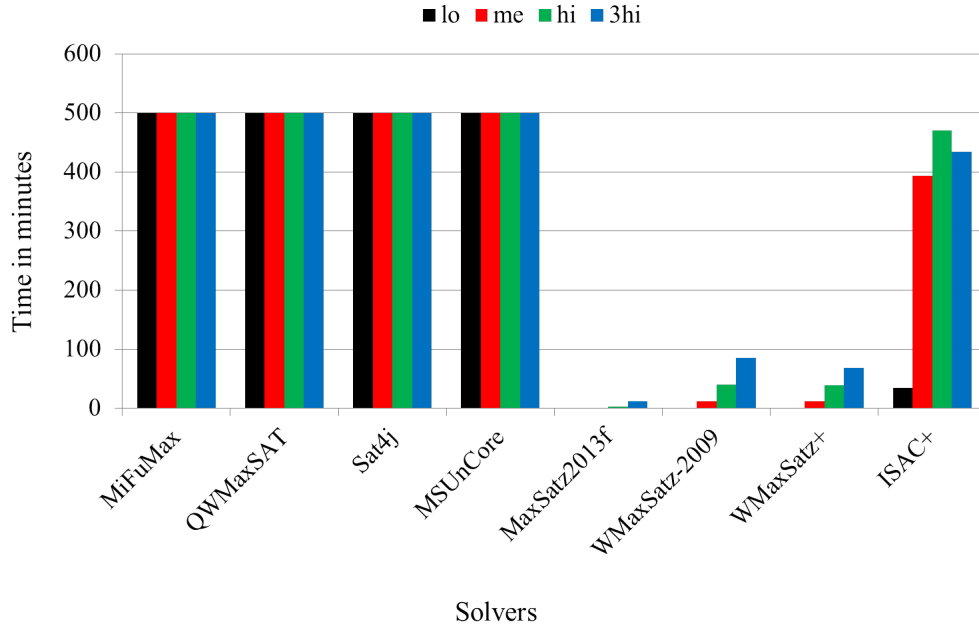


Figure 4: Time results for the random category.

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 minutes, while WMiFuMax, MSUnCore and Sat4j timedout on most instances. MaxSatz2013f placed first in the random category in the 2013 MaxSAT Evaluation, see <http://www.maxsat.udl.cat/13/results/index.html#wpms-random-pc>. The top non branch and bound solver is ISAC+, which placed third in the random category in 2014 (see <http://www.maxsat.udl.cat/14/results/index.html#wpms-random-pc>).

8.3.2 Crafted category

The seven sets of instances in the crafted category are:

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

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 3: Number of instances solved by each solver.

Solver	auc/paths	auc/sch	csg	planning	warehouses	miplib	rnd-net	Total
WMiFuMax	2.3%	100%	50%	41.1%	0%	8.3%	10.8%	30.1%
QWMaxSAT	52.3%	100%	100%	100%	11.1%	33.3%	1.4%	57%
Sat4j	31.4%	65.5%	100%	100%	5.6%	33.3%	0%	48%
MSUnCore	16.3%	100%	60%	94.6%	0%	0%	0%	38.7%
MaxSatz2013f	100%	96.4%	10%	73.2%	33.3%	33.3%	1.4%	49.7%
WMaxSatz-2009	100%	79.8%	10%	80.4%	33.3%	25%	0%	47%
WMaxSatz+	100%	78.6%	10%	80.4%	33.3%	16.7%	0%	45.6%
ISAC+	100%	100%	40%	94.6%	100%	25%	74.3%	76.3%

Table 4: Percentages of instances solved in the crafted category.

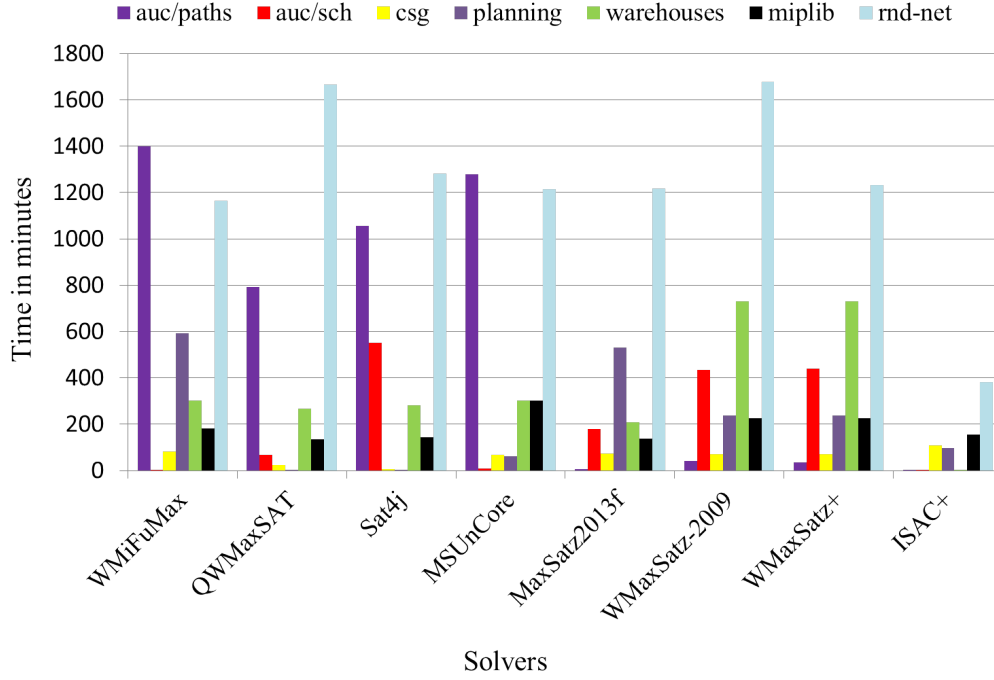


Figure 5: Time results for the crafted category.

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+ (see <http://www.maxsat.udl.cat/14/results/index.html#wpms-crafted>), and in the 2013 evaluation it placed second (see <http://www.maxsat.udl.cat/13/results/index.html#wpms-crafted-pc>). Generally, SAT-based and branch and bound solvers perform nearly equally on crafted instances.

8.3.3 Industrial category

The seven sets of instance in the industrial category are:

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

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 5: Number 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%

Table 6: Percentages of instances solved in the industrial category.

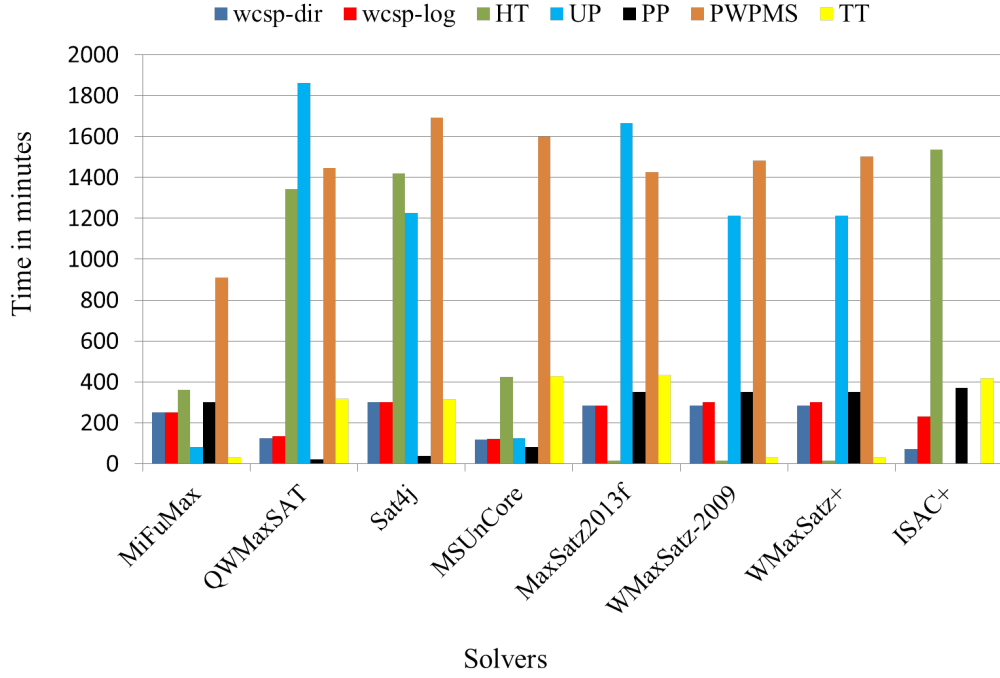


Figure 6: Time results for the industrial category.

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 evaluation is ISAC+ (see <http://www.maxsat.udl.cat/13/results/index.html#wpms-industrial>) and the same solver placed second in the 2014 evaluation (see <http://www.maxsat.udl.cat/14/results/index.html#wpms-industrial-pc>).

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.

9 Acknowledgments

This paper is made possible through the help and support from Dr. Hassan Aly (Department of Mathematics, Cairo University, Egypt) and Dr. Rasha Shaheen (Department of Mathematics, Cairo University, Egypt). I would also like to thank Dr. Carlos Ansótegui (University of Lleida, Spain) for his advice to include a section on translating pseudo Boolean constraints and his encouraging review of this work.

References

- [1] Amir Aavani. Translating pseudo-boolean constraints into cnf. *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 357–359, 2011.
- [2] Amir Aavani, David G Mitchell, and Eugenia Ternovska. New encoding for translating pseudo-boolean constraints into sat. In *SARA*, 2013.
- [3] Xuanye An, Miyuki Koshimura, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat version 0.3 & 0.4. In *Proceedings of the International Workshop on First-Order Theorem Proving, FTP*, pages 7–15, 2011.
- [4] Carlos Ansótegui, María Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 427–440, 2009.
- [5] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In *Principles and Practice of Constraint Programming*, pages 86–101. Springer, 2012.
- [6] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving wpm2 for (weighted) partial maxsat. In *Principles and Practice of Constraint Programming*, pages 117–132. Springer, 2013.
- [7] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. 2010.
- [8] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196:77–105, 2013.
- [9] Carlos Ansótegui, Yuri Malitsky, and Meinolf Sellmann. Maxsat by improved instance-specific algorithm configuration. 2014.

- [10] Josep Argelich, Chu Min Li, Felip Manyà, and Jordi Planes. The first and second max-sat evaluations. *JSAT*, 4(2-4):251–278, 2008.
- [11] Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with sat and maxsat. *Annals of Operations Research*, pages 1–21, 2012.
- [12] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming–CP 2003*, pages 108–122. Springer, 2003.
- [13] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo-boolean constraints to sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006.
- [14] Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008.
- [15] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up maxsat solving. In *Principles and Practice of Constraint Programming*, pages 247–262. Springer, 2013.
- [16] Niklas Een and Niklas Sörensson. Minisat: A sat solver with conflict-clause minimization. *Sat*, 5, 2005.
- [17] Niklas Een and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.
- [18] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. *Theory and Applications of Satisfiability Testing–SAT 2006*, pages 252–265, 2006.
- [19] Ian P Gent. Arc consistency in sat. In *ECAI*, volume 2, pages 121–125, 2002.
- [20] Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, 2004.
- [21] Federico Heras and Joao Marques-Silva. Read-once resolution for unsatisfiability-based max-sat algorithms. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence–Volume Volume One*, pages 572–577. AAAI Press, 2011.
- [22] Federico Heras, Antonio Morgado, and Joao Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In *Proceedings of the AAAI National Conference (AAAI)*, 2011.
- [23] Kazuo Iwama and Eiji Miyano. Intractability of read-once resolution. In *Structure in Complexity Theory Conference, 1995., Proceedings of Tenth Annual IEEE*, pages 29–36. IEEE, 1995.
- [24] Mikoláš Janota, Inês Lynce, Vasco Manquinho, and Joao Marques-Silva. Packup: Tools for package upgradability solving system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:89–94, 2012.

- [25] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:95–100, 2012.
- [26] Javier Larrosa, Federico Heras, and Simon de Givry. A logical approach to efficient max-sat solving. *Artificial Intelligence*, 172(2):204–233, 2008.
- [27] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
- [28] Chu Li, Felip Manyà, Nouredine Mohamedou, and Jordi Planes. Exploiting cycle structures in max-sat. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 467–480, 2009.
- [29] Chu Min Li and Anbulagan Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artificial intelligence-Volume 1*, pages 366–371. Morgan Kaufmann Publishers Inc., 1997.
- [30] Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, and Jordi Planes. Resolution-based lower bounds in maxsat. *Constraints*, 15(4):456–484, 2010.
- [31] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for max-sat. *Journal of Artificial Intelligence Research*, 30(1):321–359, 2007.
- [32] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted boolean optimization. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 495–508, 2009.
- [33] Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of pseudo-boolean constraints into cnf such that generalized arc consistency is maintained. In *KI 2014: Advances in Artificial Intelligence*, pages 123–134. Springer, 2014.
- [34] Filip Maric. Timetabling based on sat encoding: a case study, 2008.
- [35] Joao Marques-Silva. The msuncore maxsat solver. *SAT 2009 competitive events booklet: preliminary version*, page 151, 2009.
- [36] Joao Marques-Silva and Inês Lynce. Towards robust cnf encodings of cardinality constraints. *Principles and Practice of Constraint Programming-CP 2007*, pages 483–497, 2007.
- [37] Joao Marques-Silva and Jordi Planes. On using unsatisfiability for solving maximum satisfiability. *arXiv preprint arXiv:0712.1097*, 2007.
- [38] Joao Marques-Silva and Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *Proceedings of the conference on Design, automation and test in Europe*, pages 408–413. ACM, 2008.
- [39] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming*, pages 531–548. Springer, 2014.

- [40] Paulo Matos, Jordi Planes, Florian Letombe, and Joao Marques-Silva. A max-sat algorithm portfolio1. 2008.
- [41] Elizabeth Montero, María-Cristina Riff, and Leopoldo Altamirano. A pso algorithm to solve a real course+ exam timetabling problem. In *International Conference on Swarm Intelligence*, pages 24–1, 2001.
- [42] Antonio Morgado, Carmine Dodaro, and Joao Marques-Silva. Core-guided maxsat with soft cardinality constraints. In *Principles and Practice of Constraint Programming*, pages 564–573. Springer, 2014.
- [43] Antonio Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.
- [44] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.
- [45] Fahima NADER, Mouloud KOUDIL, Karima BENATCHBA, Lotfi ADMANE, Said GHAROUT, and Nacer HAMANI. Application of satisfiability algorithms to timetable problems. *Rapport Interne LMCS, INI*, 2004.
- [46] G-J Nam, Fadi Aloul, Karem A. Sakallah, and Rob A. Rutenbar. A comparative study of two boolean formulations of fpga detailed routing constraints. *Computers, IEEE Transactions on*, 53(6):688–696, 2004.
- [47] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *AAAI*, pages 2717–2723, 2014.
- [48] Steven Prestwich. Variable dependency in local search: Prevention is better than cure. In *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 107–120. Springer, 2007.
- [49] Steven David Prestwich. Cnf encodings. *Handbook of satisfiability*, 185:75–97, 2009.
- [50] Wayne Pullan. Protein structure alignment using maximum cliques and local search. In *AI 2007: Advances in Artificial Intelligence*, pages 776–780. Springer, 2007.
- [51] Sean Safarpour, Hratch Mangassarian, Andreas Veneris, Mark H Liffiton, Karem Sakallah, et al. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer Aided Design, 2007. FMCAD’07*, pages 13–19. IEEE, 2007.
- [52] Tian Sang, Paul Beame, and Henry Kautz. A dynamic approach to mpe and weighted max-sat. In *Proceedings of the 20th international joint conference on Artificial intelligence*, pages 173–179. Morgan Kaufmann Publishers Inc., 2007.
- [53] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. *Principles and Practice of Constraint Programming-CP 2005*, pages 827–831, 2005.
- [54] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

- [55] Peter Steinke and Norbert Manthey. Pbliba c++ toolkit for encoding pseudo-boolean constraints into cnf. *TU Dresden, Dresden, Germany, Technical Report*, 1:2014, 2014.
- [56] Michel Vasquez and Jin-Kao Hao. A “logic-constrained” knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Computational Optimization and Applications*, 20(2):137–157, 2001.
- [57] Joost P Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.
- [58] Hui Xu, Rob A Rutenbar, and Karem Sakallah. sub-sat: A formulation for relaxed boolean satisfiability with applications in routing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(6):814–820, 2003.
- [59] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.