Journal on Satisfiability, Boolean Modeling and Computation 9 (2015) 59-81

On Using Incremental Encodings in Unsatisfiability-based MaxSAT Solving

Ruben Martins ruben.martins@cs.ox.ac.uk
Saurabh Joshi saurabh.joshi@cs.ox.ac.uk
Department of Computer Science
University of Oxford
United Kingdom

Vasco Manquinho vmm@sat.inesc-id.pt
Inês Lynce ines@sat.inesc-id.pt
INESC-ID / Instituto Superior Técnico
Universidade de Lisboa
Portugal

Abstract

In recent years, unsatisfiability-based algorithms have become prevalent as state of the art for solving industrial instances of Maximum Satisfiability (MaxSAT). These algorithms perform a succession of unsatisfiable SAT solver calls until an optimal solution is found. In several of these algorithms, cardinality or pseudo-Boolean constraints are extended between iterations. However, in most cases, the formula provided to the SAT solver in each iteration must be rebuilt and no knowledge is reused from one iteration to the next.

This paper describes how to implement incremental unsatisfiability-based algorithms for MaxSAT. In particular, we detail and analyze our implementation of the MSU3 algorithm in the OPEN-WBO framework that performed remarkably well in the MaxSAT Evaluation of 2014. Furthermore, we also propose to extend incrementality to weighted MaxSAT through an incremental encoding of pseudo-Boolean constraints. The experimental results show that the performance of MaxSAT algorithms can be greatly improved by exploiting the learned information and maintaining the internal state of the SAT solver between iterations. Finally, the proposed incremental encodings of cardinality and pseudo-Boolean constraints are not exclusive for MaxSAT usage and can be used in other domains.

KEYWORDS: *Maximum Satisfiability, Unsatisfiability-based Algorithm, Incremental Encoding*

Submitted November 2014; revised June 2015; published December 2015

1. Introduction

The number of application domains of Maximum Satisfiability (MaxSAT) has been growing in the last few years. Application domains such as software upgradability [6], error localization in C code [28], debugging of hardware designs [14], haplotyping with pedigrees [23], and course timetabling [7] have benefited from the advancements in MaxSAT algorithms.

The success of the annual MaxSAT Evaluation has derived from a diversity of problem instances, as well as an increase of performance in MaxSAT algorithms. Each year, in

particular on industrial tracks, the solver with the most solved instances in the previous evaluation is surpassed by a new solver with better algorithmic techniques.

More recently, several unsatisfiability-based algorithms for MaxSAT have been proposed with ever better performance [22, 4, 39, 5, 24, 47, 27, 46, 50]. These algorithms are iterative in nature where a SAT solver is called in succession until an optimal solution to the MaxSAT instance is found. However, at each iteration, these algorithms create a new instance of the SAT solver and rebuild the formula losing most, if not all, of the knowledge that could be derived from previous iterations. Despite being well-known that incremental approaches have provided a huge leap in the performance of SAT solvers [55, 20, 49, 9], the notion of incrementality has not yet been fully exploited in unsatisfiability-based MaxSAT solving. One of the advantages of an incremental approach is to allow the SAT solver to retain knowledge from previous iterations that may be used in the upcoming iterations. The goal is to maintain the inner state of the SAT solver, as well as learned clauses that were discovered during the solving process of previous iterations.

In many unsatisfiability-based MaxSAT algorithms, the working formula is modified between iterations with new cardinality or pseudo-Boolean constraints [47]. These constraints are encoded in CNF so that a SAT solver can handle the resulting formula [10, 54, 21, 11, 8, 25]. In this paper we discuss the use of cardinality and pseudo-Boolean constraints in an incremental fashion in order to improve the performance of MaxSAT algorithms. Although we have proposed incremental cardinality constraints in our earlier work [43], in this paper we extend it in several ways: (1) provide an analysis on the properties of the incremental encoding, (2) propose and analyze an incremental pseudo-Boolean encoding for weighted MaxSAT and (3) provide a detailed analysis of the incremental approaches on the MaxSAT Evaluation 2014 instances.

The paper is organized as follows. Section 2 introduces preliminaries, notations, as well as the MaxSAT algorithms and encodings used in the remainder of the paper. Section 3 proposes the incremental encodings for cardinality and pseudo-Boolean constraints. Previous related work on incrementality is referred in Section 4. An extended experimental analysis on the MaxSAT Evaluation instances is provided in Section 5. Finally, the paper concludes in Section 6.

2. Preliminaries

A Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses, where a clause is a disjunction of literals. Here, a literal is a Boolean variable x_i or its negation $\neg x_i$. A Boolean variable may be assigned truth values *true* or *false*. A literal x_i ($\neg x_i$) is said to be satisfied if the respective variable is assigned value *true* (*false*) and to be unsatisfied if the respective variable is assigned value *false* (*true*). A clause is satisfied if and only if at least one of its literals is satisfied. A clause is called a unit clause if it only contains exactly one literal. A formula φ is satisfied if all of its clauses are satisfied. The Boolean Satisfiability (SAT) problem can be defined as finding a satisfying assignment to a propositional formula φ or prove that such an assignment does not exist. Throughout this paper, we will refer to φ as a set of clauses, where each clause c is a set of literals.

The Maximum Satisfiability (MaxSAT) problem is an optimization version of SAT where the goal is to find an assignment to the input variables such that the number of unsatisfied

(satisfied) clauses is minimized (maximized). In the context of this paper, it is assumed that MaxSAT is defined as a minimization problem.

MaxSAT has several variants such as partial MaxSAT, weighted MaxSAT and weighted partial MaxSAT [34]. A partial MaxSAT formula φ has the form $\varphi_h \cup \varphi_s$ where φ_h and φ_s denote the set of hard and soft clauses, respectively. The goal in partial MaxSAT is to find an assignment to the input variables such that all hard clauses φ_h are satisfied, while minimizing the number of unsatisfied soft clauses in φ_s. In the weighted versions of MaxSAT, each soft clause c_i is associated with an integer weight w_i such that $w_i \geq 1$. In this case, the goal is to find an assignment such that all hard clauses are satisfied and the total weight of unsatisfied soft clauses is minimized.

Cardinality constraints are a well-known generalization of propositional clauses. In a cardinality constraint, one encodes that at most k out of n literals can be assigned to *true*, i.e. $\sum_{i=1}^{n} l_i \leq k$ where l_i is a literal. A generalization of cardinality constraints are pseudo-Boolean constraints where each literal can have a weight, i.e. $\sum_{i=1}^{n} w_i \cdot l_i \leq k$. In this case, the weighted sum of the literals assigned to *true* must be smaller than or equal to k.

Neither cardinality nor pseudo-Boolean constraints occur in MaxSAT formulations, but their use in MaxSAT algorithms is common [22, 39, 4, 24, 47]. However, in order to iteratively use a SAT solver, most MaxSAT algorithms encode cardinality [10, 54, 8] and pseudo-Boolean constraints into CNF [56, 21, 11, 25].

2.1 Using a SAT solver

Most of the current state of the art MaxSAT solvers are based on successive calls to a SAT solver. A SAT solver call $\texttt{SAT}(\varphi, \mathcal{A})$ takes as input a CNF formula φ and a set of assumptions $\mathcal{A}$. The set of assumptions $\mathcal{A}$ defines a set of literals that must be satisfied in the model of φ returned by the solver call. Hence, the SAT call can terminate as soon as the SAT solver determines that at least one of the literals in $\mathcal{A}$ must remain unsatisfied. It is important to observe the difference between an assumption literal and a unit clause. Note that an assumption just controls the value of a variable for a given SAT call, while a unit clause defines the value of a variable for all of the calls to the SAT solver after the unit clause has been added.

The result of a SAT call is a triple (st, ν, φ_C), where st denotes the status of the solver: satisfiable (SAT) or unsatisfiable (UNSAT). If the solver returns SAT, then the model that satisfies φ is stored in ν. On the other hand, if the solver returns UNSAT, then φ_C contains an unsatisfiable formula that explains a reason for the unsatisfiability of φ [1]. Notice that φ may be satisfiable, but the solver returns UNSAT due to the set of assumptions $\mathcal{A}$ (i.e. there are no models of φ where all assumption literals are satisfied). In this case, φ_C contains a subset of clauses from φ and a subset of assumptions from $\mathcal{A}$. Otherwise, if φ is unsatisfiable, then φ_C is a subformula of φ.

For all algorithms presented in this paper it is assumed that a SAT call has been made to check the satisfiability of the set of hard clauses φ_h. If φ_h is not satisfiable, then the MaxSAT instance does not have a solution.

[1] A common approach to extract an unsatisfiable subformula is to use assumptions [19].

Algorithm 1: Linear Search Unsat-Sat Algorithm

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ

1 $(\varphi_W, V_R, \lambda) \leftarrow (\varphi_h, \emptyset, 0)$
2 **foreach** $c_i \in \varphi_s$ **do**
3 $V_R \leftarrow V_R \cup \{r_i\}$ `// ` r_i `is a new relaxation variable`
4 $c_R \leftarrow c_i \cup \{r_i\}$
5 $\varphi_W \leftarrow \varphi_W \cup \{c_R\}$
6 **while** true **do**
7 $(\mathsf{st}, \nu, \varphi_C) \leftarrow \mathtt{SAT}(\varphi_W \cup \{\mathtt{CNF}(\sum_{r_i \in V_R} w_i \cdot r_i \leq \lambda)\}, \emptyset)$
8 **if** $\mathsf{st} = \mathsf{SAT}$ **then**
9 **return** ν `// satisfying assignment to` φ
10 $\lambda \leftarrow \mathtt{UpdateBound}(\{w_i : r_i \in V_R\}, \lambda)$

2.2 MaxSAT Algorithms

The classical approach for solving MaxSAT is branch and bound algorithms using MaxSAT inference techniques and procedures to estimate the number of unsatisfied clauses to prune the search [34]. However, in recent years, state of the art MaxSAT algorithms for industrial/application instances rely on iteratively calling a SAT solver until an optimal solution is found.

One of the approaches is to perform a linear search on the number of unsatisfied soft clauses. In these algorithms, a new relaxation variable $r_i \in V_R$ is initially added to each soft clause c_i. Notice that if an original soft clause c_i is unsatisfied, then r_i must be assigned to *true*. Algorithm 1 illustrates a linear search on the total weight of unsatisfied soft clauses. At each iteration of the algorithm, a pseudo-Boolean constraint is defined such that the total weight of the unsatisfied soft clauses must be smaller than or equal to λ. The pseudo-Boolean constraint is encoded into CNF and given to the SAT solver (line 7). Observe that a pseudo-Boolean constraint is only necessary for weighted MaxSAT. If the MaxSAT formula is an instance of the partial MaxSAT problem where all soft clauses c_i have weight 1, then a cardinality constraint is used instead.

Algorithm 1 follows an Unsat-Sat linear search. At each step of the algorithm, λ defines a lower bound on the value of the optimal solution. After an unsatisfiable answer from the SAT solver, the value of λ is updated accordingly using function `UpdateBound` (line 10). If φ is a partial MaxSAT formula, then function `UpdateBound` just updates λ with $\lambda + 1$. Otherwise, if φ is a weighted MaxSAT formula, then `UpdateBound` returns the smallest integer value υ such that $\upsilon > \lambda$ and $SubSetSum(\{w_i : r_i \in V_R\}, \upsilon)$ is true [5]. Function $SubSetSum(S, \upsilon)$ solves the well-known subset sum problem, i.e. it returns true if there is a subset S' of S such that the sum of the elements of S' equals υ. Since the subset sum problem is NP-Hard, a pseudo-polynomial algorithm based on dynamic programming is used. This allows to skip over lower bound values that are not possible to attain, given the weights of the relaxed soft clauses in V_R [5].

A converse approach to Algorithm 1 is the Sat-Unsat linear search where λ is defined as an upper bound. In that case, λ is initialized with the sum of the weights of all soft clauses.

Algorithm 2: WMSU3 Algorithm

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ

1 $(\varphi_W, V_R, \lambda) \leftarrow (\varphi, \emptyset, 0)$
2 **while** true **do**
3 $\quad (\mathsf{st}, \nu, \varphi_C) \leftarrow \mathtt{SAT}(\varphi_W \cup \{\mathtt{CNF}(\sum_{r_i \in V_R} w_i \cdot r_i \leq \lambda)\}, \emptyset)$
4 $\quad$ **if** st = SAT **then**
5 $\quad\quad$ **return** ν // satisfying assignment to φ
6 $\quad$ **foreach** $c_i \in (\varphi_C \cap \varphi_s)$ **do**
7 $\quad\quad V_R \leftarrow V_R \cup \{r_i\}$ // r_i is a new variable
8 $\quad\quad c_R \leftarrow c_i \cup \{r_i\}$ // c_i was not previously relaxed
9 $\quad\quad \varphi_W \leftarrow (\varphi_W \setminus \{c_i\}) \cup \{c_R\}$
10 $\quad \lambda \leftarrow \mathtt{UpdateBound}(\{w_i : r_i \in V_R\}, \lambda)$

Next, while the SAT call returns SAT, λ is decreased. The algorithm ends when the SAT call returns UNSAT and the last found satisfying assignment is an optimal solution to φ.

Unsatisfiability-based algorithms [2.] for MaxSAT take advantage of the certificates of unsatisfiability produced by the SAT solver. Instead of relaxing all soft clauses at the beginning of the algorithm, soft clauses are relaxed only when needed. Since the Fu and Malik [22] algorithm was presented, there has been a growing diversity of unsatisfiability-based algorithms being proposed. In this paper, we focus solely on the WMSU3 algorithm [42] implemented in our tool open-wbo. This algorithm will be used to demonstrate the enhancements proposed in the paper. We refer to the literature for an extended overview on unsatisfiability-based algorithms [47].

Algorithm 2 also follows a linear search Unsat-Sat, but soft clauses are only relaxed when they appear in some unsatisfiable subformula φ_C. As in Algorithm 1, it starts with a lower bound value λ at 0 and increases its value at each iteration using `UpdateBound` (line 10). When the working formula becomes SAT, the algorithm ends. Otherwise, at each unsatisfiable call on the working formula, non relaxed soft clauses that appear in φ_C are relaxed (lines 6-9). As a result, besides the lower bound value λ, the left hand side of the pseudo-Boolean constraint can also be modified at each iteration (line 3). However, one should note that in the WMSU3 algorithm, some soft clauses might never have to be relaxed. Therefore, the constraint size on the lower bound value λ is usually smaller than in Algorithm 1.

Note that the MSU3 algorithm is a special case of WMSU3 for partial MaxSAT instances. In this case, the weight of soft clauses is always 1. Therefore, one only needs to encode a cardinality constraint in line 3. Moreover, the increase of the lower bound λ at each iteration is always 1 (line 10).

2.3 Totalizer Encoding

As previously shown, cardinality constraints can occur when solving MaxSAT formulations. In the presented algorithms, this happens when we have a partial MaxSAT formula. In this case, the cardinality constraint must be encoded into CNF in order for the SAT solver to be

[2.] Also known as core-guided algorithms [47].

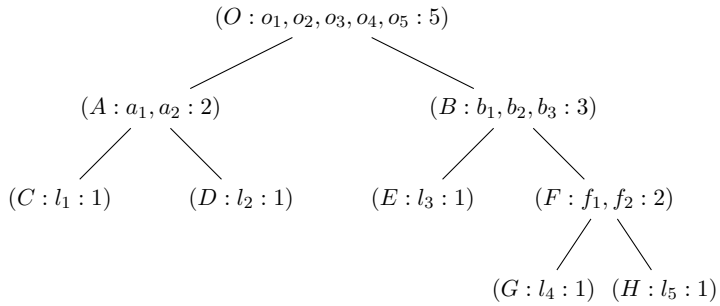


Figure 1: Totalizer encoding for $l_1 + \cdots + l_5 \leq k$

able to deal with it. For the purpose of this paper, we describe the Totalizer encoding [10] for cardinality constraints, as later in the paper we build upon this encoding to present *iterative encoding*. However, other effective alternative encodings for cardinality constraints exist [54, 8, 51].

The Totalizer encoding can be visualized as a tree as shown in Fig. 1. Here, the notation for every node is $(node\_name : node\_vars : node\_sum)$. To enforce the cardinality constraint, we need to count how many input literals $(l_1, \ldots, l_n)$ are set to *true*. This counting is done via a unary representation. Therefore, at every node its corresponding $node\_vars$ represent integers from 1 to $node\_sum$ in ascending order. For example, at node B, b_2 being set to *true* means that at least two of the leaves under the tree rooted at B have been set to *true*. The input literals $(l_1, \ldots, l_5)$ are at the leaves whereas the root node has the output variables $(o_1, \ldots, o_5)$ giving the final tally of how many input literals have been set.

Any intermediate node P, counting up to n_1, has two children Q and R counting up to n_2 and n_3 respectively such that $n_2 + n_3 = n_1$. Also, their corresponding $node\_vars$ will be $(p_1, \ldots, p_{n_1})$, $(q_1, \ldots, q_{n_2})$ and $(r_1, \ldots, r_{n_3})$ in that order. In order to ensure that the correct sum is received at P, the following formula is built for P:

$$\bigwedge_{\substack{0 \leq \alpha \leq n_2 \\ 0 \leq \beta \leq n_3 \\ 0 \leq \sigma \leq n_1 \\ \alpha + \beta = \sigma}} \neg q_\alpha \vee \neg r_\beta \vee p_\sigma \quad \text{where, } p_0 = q_0 = r_0 = 1 \tag{1}$$

Essentially, (1) dictates that if α many leaves have been set to *true* under the subtree rooted at Q and β many leaves have been set to *true* under the subtree rooted at R then p_σ must be set to *true* to indicate that at least $\alpha + \beta$ many leaves have been set to *true* under P. Observe that (1) only counts the number of input literals set to *true*. In other words, it encodes the *cardinality sum* over the input literals. To enforce that at most k of the input literals are set to *true*, we conjunct it with the following :

$$\bigwedge_{k+1 \leq i \leq n} \neg o_i \tag{2}$$

Observation 1 *Two disjoint subtrees for the Totalizer encoding are independent of each other. For example, the tree rooted at B counts how many literals have been set to true from $\{l_3, l_4, l_5\}$ whereas, the tree rooted at A counts the literals that are set to true from $\{l_1, l_2\}$.*

Note also that (1) counts up to n and then (2) restricts the sum to k. If we only want to enforce the constraint for at most k then we need at most $k+1$ output variables at the root. In turn, we need at most $k+1$ *node_vars* at any intermediate node. Even with this modification, the formula in (1) remains valid. However, the equality $n_2 + n_3 = n_1$ may no longer hold. With this modification, (2) simplifies to

$$\neg o_{k+1}$$

Without the simplification this encoding requires $O(nlog\,n)$ extra variables and $O(n^2)$ clauses. After the simplification the number of clauses reduces to $O(nk)$ [13, 31]. From here on, we will refer to this simplification as *k-simplification*.

Observation 2 *Let φ_1 and φ_2 be two formulas, representing cardinality sums k_1 and k_2 respectively, generated using (1) and k-simplification. Observe that $\varphi_1 \subset \varphi_2$, whenever $k_1 < k_2$.*

2.4 Sequential Encoding

In this section, we briefly describe the sequential weight counter (SWC) encoding [25] of pseudo-Boolean (PB) constraints into CNF. As with the cardinality encoding presented in the previous section, we will also build on the SWC encoding for presenting our approach for solving weighted MaxSAT formulas. Nevertheless, we note that this is only one of many alternative encodings for pseudo-Boolean constraints into CNF [56, 21, 11].

The goal of the SWC encoding is to encode into CNF a PB constraint of the form $\sum_{i=i}^{n} w_i x_i \leq k$. In the SWC encoding, a new set of variables $s_{i,j}$ is defined, where $1 \leq i < n$ and $1 \leq j \leq k$. Variable $s_{i,j}$ must be assigned value 1 if the weighted sum of the first i literals is larger than or equal to j. Otherwise, $s_{i,j}$ is assigned value 0.

Suppose we have the constraint $2x_1 + 3x_2 + 3x_3 \leq 5$ and $x_1 = x_3 = 1$ and $x_2 = 0$. In this case, $s_{2,4}$ can be assigned value 0 since the weighted sum of the first two literals is smaller than 4. However, since the third literal x_3 is assigned value 1, then $s_{3,4}$ must be assigned value 1.

The SWC encoding [25] is as follows:

$$(\neg s_{i-1,j} \vee s_{i,j}) \quad \text{for} \quad 2 \leq i < n, 1 \leq j \leq k \tag{3}$$

$$(\neg x_i \vee s_{i,j}) \quad \text{for} \quad 1 \leq i < n, 1 \leq j \leq w_i \tag{4}$$

$$(\neg s_{i-1,j} \vee \neg x_i \vee s_{i,j+w_i}) \quad \text{for} \quad 2 \leq i < n, 1 \leq j \leq k - w_i \tag{5}$$

$$(\neg s_{i-1,k+1-w_i} \vee \neg x_i) \quad \text{for} \quad 2 \leq i \leq n \tag{6}$$

The clause set (3) ensures that if the weighted sum of the first $i-1$ literals is at least j, then it is also at least j if we consider the first i literals. This is true since the weights are all non-negative. Next, clause set (4) ensures that if literal x_i is assigned value 1, then the weighted sum considering the first i literals is at least w_i. Hence, all $s_{i,j}$ where $j \leq w_i$ are assigned value 1 if $x_i = 1$. Clause set (5) follows a similar reasoning. If the weighted sum of the first $i-1$ literals is at least j and x_i is assigned value 1, then the weighted sum of the first i literals is at least $j + w_i$. Finally, clause set (6) limits the weighted sum of literals up to k. If the weighted sum of the first $i-1$ literals is at least $k+1-w_i$, then x_i must be assigned value 0. Otherwise, the weighted sum of the literals would be larger than k and the original PB constraint would not be satisfied.

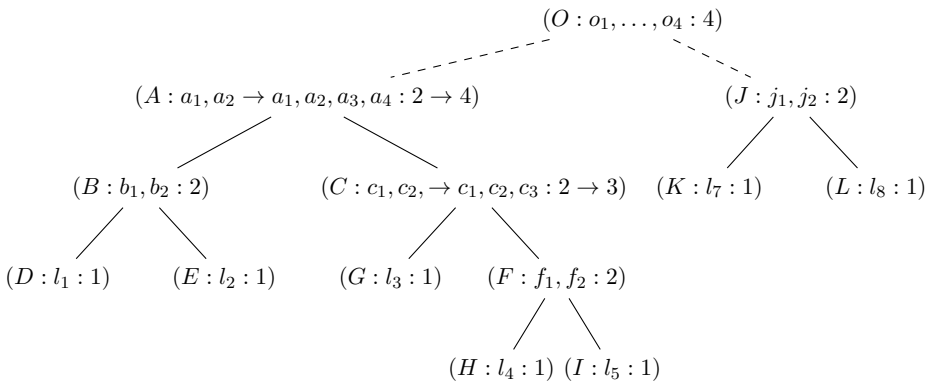


Figure 2: Transforming $l_1 + \cdots + l_5 \leq 1$ and $l_7 + l_8 \leq 1$ into $l_1 + \ldots + l_5 + l_7 + l_8 \leq 3$

3. Iterative incrementality

Unsatisfiability-based algorithms for MaxSAT are based on relaxing unsatisfiable formulas until a satisfiable formula is reached, such that the corresponding model is an optimal solution of the MaxSAT instance. Until recently, unsatisfiability-based algorithms were non-incremental, i.e. at each iteration of the algorithm, the SAT solver was rebuilt with a new CNF instance to be solved. In this section we review the iterative encoding [43] of cardinality constraints that allows an incremental implementation of the MSU3 algorithm for partial MaxSAT instances. This incremental scheme has allowed a very simple unsatisfiability-based algorithm to be the best non-portfolio solver for industrial unweighted and industrial partial MaxSAT instances in the MaxSAT Evaluation 2014[3]. Furthermore, in this section, we propose to extend the iterative encoding to pseudo-Boolean constraints, thus allowing our implementation of WMSU3 to be the first fully incremental algorithm for weighted MaxSAT.

3.1 Iterative Cardinality Encoding

One approach to implement incrementality is to use incremental weakening [43]. The main idea is to use a conservative upper bound k_u. In partial MaxSAT, k_u would be an upper bound on the number of unsatisfied soft clauses that could be defined by finding any model to the hard clause set. In this case, the cardinality constraint that limits the number of relaxation variables to be assigned value 1 would be encoded only once with k_u on the right hand side. However, at each iteration of the algorithm, one would limit the effective output of the left hand side of the cardinality constraint by using assumptions [43].

There are two main drawbacks with this approach: (i) all soft clauses would have to be initially relaxed and (ii) the conservative upper bound k_u can be much larger than the optimum value k_{opt}. As a result, the size of the encoding of the cardinality constraint can be much larger than the one used in the non-incremental approach.

[3] Results available at http://www.maxsat.udl.cat/14/.

Since incremental weakening does not allow the set of input literals in the cardinality constraint to change, MaxSAT algorithms that only relax soft clauses when they appear in some unsatisfiable subformula can not take full advantage of incremental weakening.

To remedy this situation, we propose to encode the cardinality constraint in an iterative fashion. At each iteration of the MaxSAT algorithm, the encoding of the cardinality constraint is augmented with clauses that allow the sum of the input literals to go up to k for the current iteration. We call this approach *iterative encoding*.

Let us take a look at Fig. 2 to see how iterative encoding proceeds. Assume that for a particular iteration, we needed to encode $l_1 + \cdots + l_5 \leq 1$. This can be accomplished using the subtree rooted at A. Since the bound for this iteration is $k = 1$, we only need $k + 1 = 2$, *node_vars* at every node as described in k-simplification in Section 2.3. In the next iteration, suppose we need to encode $l_1 + \cdots + l_5 + l_7 + l_8 \leq 3$. Observation 2 allows us to augment the formula for subtree rooted at A to allow $l_1 + \cdots + l_5$ to sum up to 4. This is done by increasing the output variables of node A to sum up to 4 and adding the respective clauses that encode sums 3 and 4. Similarly, for node C the output variables are increased to sum up to 3 and the clauses that sum up to 3 are added to the formula. For the additional input literals l_7 and l_8 we encode the subtree rooted at J. Observation 1 allows us to merge trees rooted at A and J by creating a new parent node O which sums up to 4 since A and J have disjoint sets of input literals. To restrict the number of input literals being set to *true* to 3, we only need to add $\neg o_4$ as described in (2).

In general, if the cardinality constraint changes from $x_1 + \cdots + x_n \leq k_1$ $(k_1 < n)$ to $x_1 + \cdots + x_n + y_1 + \cdots + y_m \leq k_2$ where $k_1 \leq k_2$ then we do the following 4 steps:

1. Remove the assumption over output literal $\neg o_{k_1+1}$ which restricts the sum of $x_1 \ldots, x_n$ to k_1.

2. Augment the formula for $x_1, \ldots, x_n$ to sum up to $min(k_2 + 1, n)$.

3. Encode the formula over $y_1, \ldots, y_m$ to sum up to $min(k_2 + 1, m)$.

4. Conjunct these two formulas and augment the resulting formula using (1) and k-simplification in order to encode $x_1 + \cdots + x_n + y_1 + \cdots + y_m \leq k_2$.

Since the iterative encoding always adds clauses to the existing formula and changes assumptions, it allows us to retain the internal state of the solver across iterations.

For partial MaxSAT instances, a linear search Unsat-Sat algorithm (Algorithm 1), increases the lower bound by 1 at each iteration but does not change the set of input literals of the cardinality constraint. Therefore, to apply iterative encoding to this algorithm we only have to perform steps 1 and 2. On the other hand, MSU3 algorithm (Algorithm 2 for partial MaxSAT instances) may change the set of input literals of the cardinality constraint between iterations. Therefore, iterative encoding is applied to MSU3 by performing steps 1 to 4.

Since at every iteration, a bare minimum number of clauses necessary to encode the cardinality constraint for that iteration is added, the size of the encoding remains small throughout the run of the MaxSAT algorithm. The iterative encoding is not only faster but allows us to solve more problem instances as compared to non-incremental approaches.

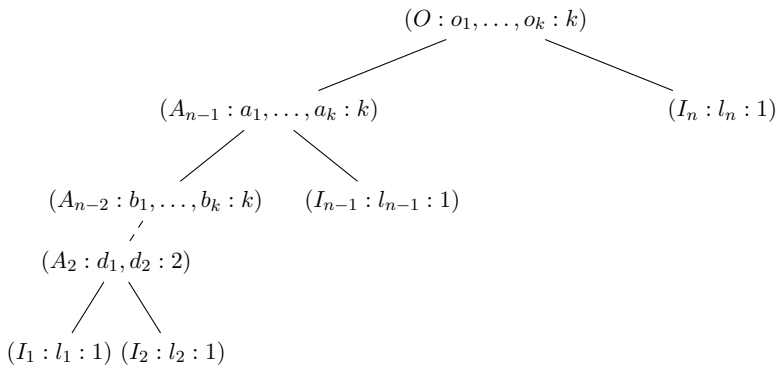


Figure 3: Worst-case Totalizer tree encoding for $l_1 + \cdots + l_n < k$

Encoding Properties

The Totalizer encoding needs $O(n\,log_2 n)$ additional variables and $O(nk)$ clauses [10]. This complexity is achieved assuming the tree built for the encoding is balanced. However, the iterative Totalizer encoding described in this section does not guarantee a balanced tree for the structure of the encoding. At each iteration, a new root node is created with the existing tree as one child and a new tree branch (which encodes the additional set of input literals) as the other child. In the worst case, at every iteration only one new input literal has to be considered, resulting in a completely skewed tree as shown in Fig. 3.

The total number of nodes in a tree which encodes n input literals would be $O(n)$. Since at every intermediate node, only k many *node_vars* are required, the iterative encoding requires $O(nk)$ encoding variables. At every intermediate node, $O(k^2)$ clauses are required to encode the cardinality sum. Thus, in the worst case, the total number of clauses is $O(nk^2)$. However, such an extreme case of completely unbalanced tree does not seem to occur in practice.

Note that the arc-consistency property of the Totalizer encoding still holds for the iterative version. Since the arc-consistency proof provided in the original paper does not depend on the encoding tree being balanced, here we refer to the original proof [10]. The proof is based on an induction step, but for it to go through, it is sufficient for the left and right child to have strictly less variables than the parent node. Hence, the n input variables can be split into 1 and $n-1$ variables (or any other splitting procedure) and the proof still holds.

3.2 Iterative Pseudo-Boolean Encoding

This section describes how to perform the incremental encoding of a PB constraint using the SWC encoding described in section 2.4.

Consider that a PB constraint defined over n literals $\sum_{i=1}^{n} w_i x_i \leq k$ has already been encoded into CNF and added to the SAT solver. The main goal is to be able to extend the CNF encoding of the original PB constraint to represent a new PB constraint $\sum_{i=1}^{m} w_i x_i \leq k'$ where $m \geq n$ and $k' \geq k$. Hence, in the iterative encoding we consider adding new weighted

literals to the left hand side (the number of literals can increase to m) and increasing the right hand side of the PB constraint from k to k'.

First we analyze how to encode an increase on the right hand side of the PB constraint while maintaining the left hand side. We start by verifying which part of the initial encoding remains valid with an increase of the right hand side value from k to k'. Clearly, clauses defined in (3) to (5) are also valid with the new right hand side value k'. However, this is not the case for clause set (6). Hence, in order to be able to perform an incremental encoding, we propose to change (6) by adding a blocking variable b_k. As a result, clause set (6) is replaced by:

$$(b_k \vee \neg s_{i-1,k+1-w_i} \vee \neg x_i) \qquad \text{for} \quad 2 \leq i \leq n \tag{7}$$

Therefore, in the SAT solver call, blocking literal $\neg b_k$ must be included as an assumption, in order to guarantee that the weighted sum of literals is limited to k.

The incremental step to increase the right hand side of the PB constraint from k to k' can then be done by adding the following clauses:

$$(\neg s_{i-1,j} \vee s_{i,j}) \qquad \text{for} \quad 2 \leq i < n, k < j \leq k' \tag{8}$$

$$(\neg s_{i-1,j} \vee \neg x_i \vee s_{i,j+w_i}) \qquad \text{for} \quad 2 \leq i < n, k - w_i < j \leq k' - w_i \tag{9}$$

$$(b_{k'} \vee \neg s_{i-1,k'+1-w_i} \vee \neg x_i) \qquad \text{for} \quad 2 \leq i \leq n \tag{10}$$

$$(b_k) \tag{11}$$

Note that (8) and (9) correspond to expanding (3) and (5) up to k'. Notice also that (11) disables all clauses in (7), thus removing the limit of k to the encoded PB constraint. Finally, (10) introduces a new blocking variable $b_{k'}$ and establishes the new limit of k' to the right hand side. In the next SAT solver call, $\neg b_k$ must be replaced by $\neg b_{k'}$ in the assumption set.

Now we consider adding new weighted literals to the left hand side, while keeping the right hand side of the PB constraint k'. The goal is to extend the encoding of PB constraint $\sum_{i=1}^{n} w_i x_i \leq k'$ defined over a set of n weighted literals to a new PB constraint $\sum_{i=1}^{n} w_i x_i + \sum_{i=n+1}^{m} w_i x_i \leq k'$ defined over a set of m weighted literals. In this case, one just needs to extend the original SWC encoding (3)-(6) to consider the new literals from $n+1$ to m as follows:

$$(\neg s_{i-1,j} \vee s_{i,j}) \qquad \text{for} \quad n \leq i < m, 1 \leq j \leq k' \tag{12}$$

$$(\neg x_i \vee s_{i,j}) \qquad \text{for} \quad n \leq i < m, 1 \leq j \leq w_i \tag{13}$$

$$(\neg s_{i-1,j} \vee \neg x_i \vee s_{i,j+w_i}) \qquad \text{for} \quad n \leq i < m, 1 \leq j \leq k' - w_i \tag{14}$$

$$(b_{k'} \vee \neg s_{i-1,k'+1-w_i} \vee \neg x_i) \qquad \text{for} \quad n < i \leq m \tag{15}$$

As a result of these steps, it is possible to iteratively encode a PB constraint without having to rebuild the formula in the SAT solver between calls in the WMSU3 algorithm. Clearly, this approach can also be applied to different MaxSAT or other optimization algorithms using iterative calls to a SAT solver.

Encoding Properties

In order to encode a PB constraint $\sum_{i=1}^{n} w_i x_i \leq k$, the iterative encoding proposed in this section generates almost the same CNF formula as in the non-incremental case for the same iteration. The original encoding uses $O(nk)$ additional variables [25]. In our case, a new blocking variable is introduced whenever the right hand side of the PB constraint increases. Hence, in the worst case, the right hand side is increased only by 1 in each iteration up to k. Therefore, at most k blocking variables are used on the iterative encoding. As a result, the proposed iterative encoding also uses $O(nk)$ additional variables.

The original SWC encoding uses $O(nk)$ clauses [25]. Our proposed iterative encoding generates the same clauses as the original, except for the additional clauses added when the right hand side increases. In particular, clause sets (10), (11) and (15) must be added when a new blocking clause is created. Observe that the size of these clause sets is limited to $O(n)$ clauses. Considering that we have at most k blocking variables, the additional number of clauses due to (10), (11) and (15) is $O(nk)$ over all iterations used to build the PB constraint. Hence, the overall number of clauses used in our iterative encoding is maintained at $O(nk)$.

Notice that at each iteration, the blocking variable from the previous iteration is assigned to *true* (11). Therefore, the only blocking variable that remains unassigned is $b_{k'}$. However, since we have $\neg b_{k'}$ in the assumption set of the SAT solver call, the active clauses in each SAT call are the same as in the original encoding. As a result, the property of generalized arc-consistency by unit propagation also holds for the iterative encoding [25].

4. Related Work

The first use of incremental SAT solving can be traced back to the 90's with the seminal work of John Hooker [26]. Initially, only a subset of constraints is considered. At each iteration, more constraints are added to the formula. Later, incremental approaches were adopted by constraint solvers in the context of SAT [57, 19] and SAT extensions [31, 9].

Assumptions are widely used for incremental SAT [20, 49]. The minisat solver [19] interface allows the definition of a set of assumptions. Alternatively, the interface of zchaff [37] allows removing groups of clauses.

Although not implemented, the work of Fu and Malik in MaxSAT [22] discusses how learned clauses may be kept from one SAT iteration to the next one. A technique called *incremental blocking* was described and used with Fu-Malik algorithm in [43]. In incremental blocking, at each iteration i, a new auxiliary variable b_i, known as *blocking variable*, is added as a positive literal to every clause in the encoding of the cardinality constraints. In addition, $\neg b_i$ is added as an assumption to enforce the cardinality constraints. In the next iteration $i+1$, when the cardinality constraints from the previous iteration are not required, b_i is added as a unit clause to disable these constraints. In Pseudo-Boolean Optimization (PBO), early implementations include the use of incremental strengthening in minisat+ [21]. Linear search Sat-Unsat algorithms [31, 33] are implemented incrementally. A critical issue is on keeping *safe* learned clauses in successive iterations of a core-guided algorithm [44]. Quantified Boolean Formula (QBF) solving has successfully been made incremental [36] and further applied to verification [40].

In the context of SAT, incremental approaches exist for building encodings and identifying Minimal Unsatisfiable Subformulas (MUSes). For example, an incremental translation to CNF uses unit clauses to simplify the pseudo-Boolean constraint before translating it to CNF [38]. More recent work *lazily* decomposes complex constraints into a set of clauses [1]. The identification of MUSes has been made incremental by Liffton *et al.* [35]. Later on, the SAT solver Glucose has been made incremental using assumptions and applied to MUS extraction [9].

Incrementality is also present in other SAT-related domains such as Satisfiability Modulo Theories (SMT), Bounded Model Checking (BMC) and Constraint Satisfaction Problems (CSPs). The SMT-LIB v2.0 [12] defines the operations *push* and *pop* to work with a stack containing a set of formulas to be jointly solved. The MaxSAT solver WPM [3] uses the SMT solver Yices [18] which supports incrementality. The use of SAT solvers in BMC is known to benefit from incrementality, either by implementing incremental SAT solving [55] or by using assumptions [20]. Incrementality is naturally present in Dynamic CSPs (DCSPs) [17] where the formulation of a problem evolves over time by adding and/or removing variables and constraints. *Nogoods* can eventually be carried from one formulation to the next one. DCSPs make use of an incremental arc consistency algorithm [16].

Recent independent work has addressed the incremental encoding of cardinality constraints in SAT-based MaxSAT solvers [53]. Instead of translating the entire cardinality constraint into CNF, a divide-and-conquer approach is used to encode partial cardinality constraints successively. The resulting sub-problems are solved and merged incrementally, reusing not only intermediate local optima, but also additional constraints which are derived from solving the individual sub-problems by the SAT solver. In contrast to their work, we support incrementality for unsatisfiability-based algorithms.

5. Experimental Results

All of our experiments have been performed on the benchmarks taken from the industrial category of the MaxSAT Evaluation 2014. The evaluation was performed on two AMD Opteron 6276 processors (2.3 GHz) running Fedora 18 with a timeout of 1,800 seconds and memory limit of 8 GB. We implemented the incremental and non-incremental encodings for the Linear Search Unsat-Sat (LinearUS) and WMSU3 algorithms on top of open-wbo [45]. open-wbo is a modular open source MaxSAT solver that is publicly available at `http://sat.inesc-id.pt/open-wbo/`. When considering only single engine solvers, open-wbo with the incremental WMSU3 algorithm placed first[4.] in the unweighted and partial industrial MaxSAT categories at the MaxSAT Evaluation 2014.

5.1 Unweighted MaxSAT

In this section we report our results on 55 industrial unweighted MaxSAT benchmarks and on the 568 industrial partial MaxSAT benchmarks from the MaxSAT Evaluation 2014. From here on, we will use the term *unweighted* to refer to both unweighted and partial MaxSAT instances. We compare the performance of LinearUS and MSU3 against Eva [50], MSCG [39,

[4.] Version open-wbo-in in the MaxSAT Evaluation 2014.

Table 1: Number of unweighed industrial instances solved by each solver

Benchmark	#Inst	LinearUS	Inc-LinearUS	MSU3	Inc-MSU3	Eva	MSCG	Clasp	WPM
debugging	3	1	1	**3**	**3**	**3**	**3**	**3**	**3**
safarpour	52	6	2	36	**42**	38	40	39	29
aes	7	0	0	0	**1**	**1**	**1**	**1**	0
mesat	18	**11**	**11**	**11**	**11**	**11**	**11**	**11**	**11**
sugar	19	11	**12**	11	**12**	11	11	**12**	**12**
fir	32	28	28	28	**32**	29	**32**	**32**	26
simp	10	**9**	**9**	**9**	**9**	**9**	8	**9**	**9**
su	38	34	34	**35**	**35**	34	33	31	32
msp	40	8	11	9	9	**19**	11	9	11
mtg	30	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
syn	38	6	**7**	7	15	**18**	17	16	7
circuit	4	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
close_solutions	50	28	32	46	48	48	**49**	**49**	46
des	50	35	33	36	41	41	42	**45**	35
haplotype	6	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
hs-timetabling	2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
mbd	46	39	40	35	**45**	42	43	40	34
packup-pms	40	26	**40**	**40**	**40**	**40**	**40**	**40**	**40**
mqc-nencdr	25	**25**	**25**	**25**	**25**	**25**	**25**	**25**	23
mqc-nlogencdr	25	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
routing	15	**15**	**15**	**15**	**15**	**15**	**15**	**15**	**15**
protein_ins	12	**12**	**12**	**12**	**12**	8	**12**	**12**	**12**
Multiple_path	36	30	30	**36**	35	32	30	32	35
One_path	25	**25**	**25**	**25**	**25**	**25**	**25**	**25**	**25**
Total	623	414	432	484	**520**	514	513	511	470

27, 46], Clasp [2], and WPM [4, 3]. These solvers have been selected for comparison because they have shown remarkable performance[5] in the MaxSAT Evaluation 2014.

Table 1 shows the number of instances solved by each solver (*#Inst*). The `debugging` and `safarpour` benchmarks only contain soft clauses, while the remaining ones contain both soft and hard clauses. The columns *LinearUS* and *MSU3* indicate OPEN-WBO with non-incremental encodings using the unweighted version of these algorithms. The columns *Inc-LinearUS* and *Inc-MSU3* correspond to the incremental version of each algorithm. Since the information in the SAT solver is preserved across iterations, both Inc-LinearUS and Inc-MSU3 are able to solve more instances than their non-incremental counterparts. Inc-LinearUS solves 18 more instances than LinearUS, whereas Inc-MSU3 solves 36 more instances than MSU3. LinearUS is not competitive with other state-of-the-art MaxSAT solvers, and even though Inc-LinearUS improves its performance it is not sufficient to close the gap between Inc-LinearUS and other MaxSAT solvers. On the other hand, MSU3 is more competitive than LinearUS and the incremental nature of Inc-MSU3 allows it to be competitive with Eva, MSCG, and Clasp.

Fig. 4 shows a cactus plot with the running times of the solvers for unweighted MaxSAT. The gap in performance between LinearUS and Inc-LinearUS and between MSU3 and Inc-MSU3 is clear in the cactus plot. The iterative Totalizer encoding does not only significantly increase the number of solved instances but it also decreases the running time of the solvers. Due to the iterative encoding, MSU3 becomes a state-of-the-art MaxSAT algorithm for unweighted instances.

[5] Only single engine solvers have been considered in this evaluation, therefore we did not include ISAC+ (a portfolio MaxSAT solver) [30].

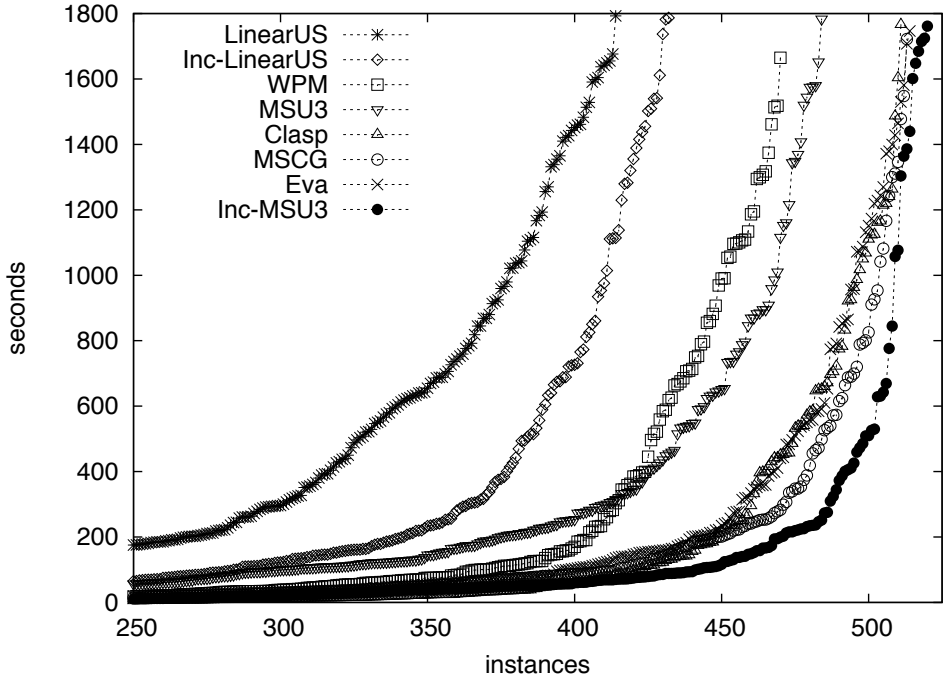


Figure 4: Running times of MaxSAT solvers for unweighted MaxSAT instances

Impact of Incrementality

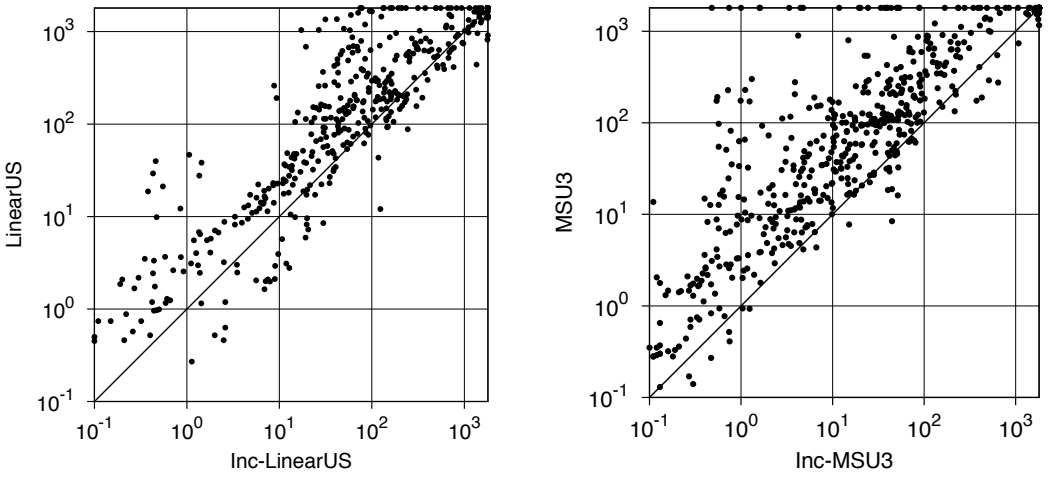
The scatter plot in Fig. 5 compares the impact of the iterative Totalizer encoding on the LinearUS and MSU3 algorithms. Each point in the plot corresponds to a problem instance, where the y-axis corresponds to the run time required by the non-incremental solver and the x-axis corresponds to the run time required by the incremental solver. Instances that are below the diagonal are solved faster when using the non-incremental solver, whereas instances that are above the diagonal are solved faster when using the incremental solver. For instances solved by both LinearUS and Inc-LinearUS, Inc-LinearUS is on average 2× faster than LinearUS. A similar pattern can be observed for Inc-MSU3, where for instances solved by both MSU3 and Inc-MSU3, Inc-MSU3 is on average 3× faster than MSU3.

The only difference between the incremental and non-incremental solvers is the incremental nature of the underlying cardinality encoding. The iterative Totalizer encoding is clearly the key ingredient in the performance boost of Inc-LinearUS and Inc-MSU3.

5.2 Weighted MaxSAT

For weighted MaxSAT, we perform our evaluation on the 410 industrial benchmarks from the weighted partial MaxSAT category of the MaxSAT Evaluation 2014. We compare the performance of LinearUS and WMSU3 against the MaxSAT solvers used in the previous section.

The number of instances solved by each solver is given in Table 2. The `pedigrees` and `upgradeability` benchmarks correspond to instances using a lexicographical optimization criterion [41]. Solving these instances can be reduced to solving a sequence of unweighted



(a) Run times of LinearUS and Inc-LinearUS (b) Run times of MSU3 and Inc-MSU3

Figure 5: Impact of the iterative Totalizer encoding

Table 2: Number of weighted industrial instances solved by each solver

Benchmark	#Inst	LinearUS	Inc-LinearUS	WMSU3	Inc-WMSU3	Eva	MSCG	Clasp	WPM
pedigrees	100	64	98	91	99	**100**	**100**	90	94
upgradeability	100	31	66	99	**100**	**100**	**100**	**100**	**100**
hs-timetabling	14	1	1	2	2	1	1	**3**	**3**
packup-wpms	99	0	0	0	0	**99**	**99**	95	93
planning	29	28	**29**	28	**29**	**29**	28	27	**29**
timetabling	26	7	7	6	10	11	**13**	11	**13**
spot5-dir	21	4	8	6	7	14	14	13	**17**
spot5-log	21	4	7	5	6	**14**	10	**14**	11
Total	410	139	216	237	253	**368**	365	353	360

partial MaxSAT instances. For such instances, we used the iterative Totalizer encoding. For the remaining instances, we used the iterative SWC encoding. The columns *LinearUS* and *WMSU3* indicate open-wbo with non-incremental encodings using the LinearUS and WMSU3 algorithms, respectively. The columns *Inc-LinearUS* and *Inc-WMSU3* indicate the incremental version of each algorithm. Since the iterative Totalizer encoding can be used for `pedigrees` and `upgradeability`, the performance of Inc-LinearUS is significantly improved and the performance of Inc-WMSU3 is comparable to other solvers. Incrementality results in Inc-LinearUS solving 69 more instances as compared to LinearUS and Inc-WMSU3 solving 9 more instances as compared to WMSU3 on these two benchmarks. LinearUS and WMSU3 are unable to solve any instance in the `packup-wpms` benchmark set. These instances have large optimum values (in the order of hundreds of thousands). The lower bound increment performed by LinearUS and WMSU3 is too small to handle such instances. Since the lower bound increment strategy is the same for the incremental encoding, Inc-LinearUS and Inc-WMSU3 also do not solve any instance of this benchmark set. For the remaining instances, the iterative SWC encoding helps Inc-LinearUS to solve 8 more instances than LinearUS and Inc-WMSU3 to solve 7 more instances than WMSU3. The table highlights the fact

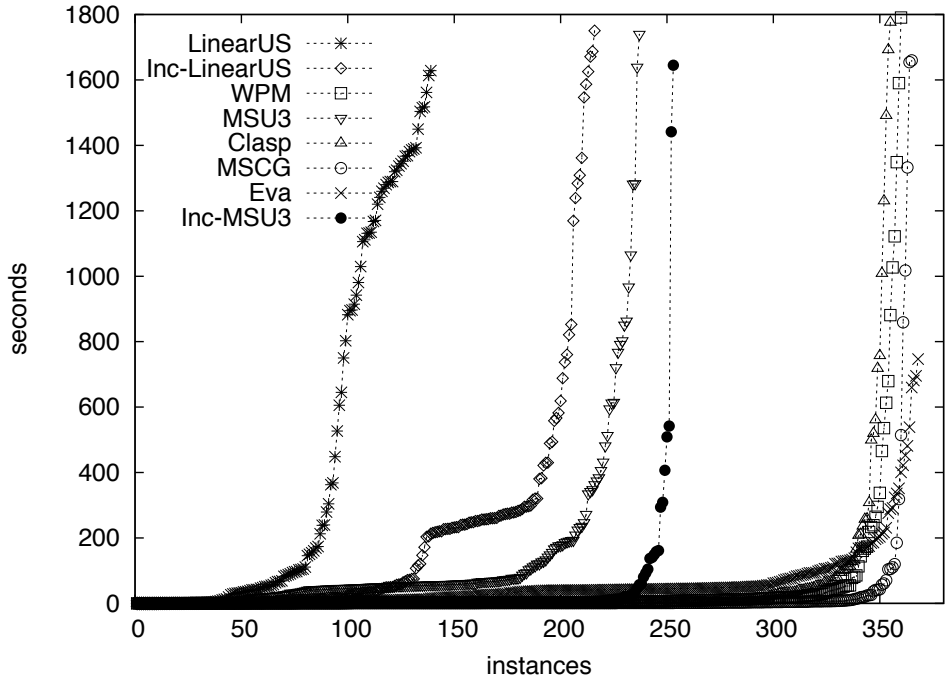


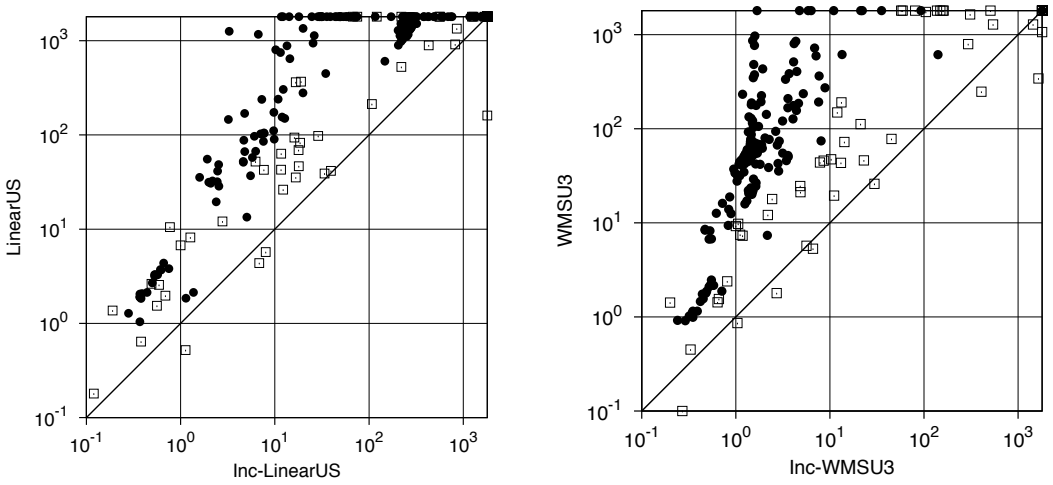
Figure 6: Running times of MaxSAT solvers for weighted MaxSAT instances

that the iterative encoding (whether used with Totalizer or SWC) always improves the performance of the underlying MaxSAT algorithm.

Fig. 6 compares the performance of the solvers for weighted MaxSAT. The performance of LinearUS, Inc-LinearUS, WMSU3 and Inc-WMSU3 is not comparable to the remaining solvers. However, note that the iterative encoding is not specific to a particular MaxSAT algorithm. We believe that the iterative encoding can improve the efficiency of any algorithm that uses cardinality or pseudo-Boolean constraints. For example, MSCG [46] could be enhanced with the iterative encoding. In the case of Inc-LinearUS and Inc-WMSU3, we can see in the cactus plot that the iterative encoding does enhance the performance.

Impact of Incrementality

Fig. 7 shows a scatter plot that compares the impact of the incremental encoding on the WMSU3 algorithm for weighted benchmarks. Instances with lexicographical optimization criterion are represented as circles, while the remaining ones are shown as squares. Inc-LinearUS dominates LinearUS with an average speedup of 10 for instances where the iterative Totalizer encoding is used. For the remaining ones, Inc-LinearUS outperforms LinearUS by a factor of 2. A similar pattern can be observed with Inc-WMSU3. Inc-WMSU3 dominates WMSU3 with an average speedup of 31 for instances where the iterative Totalizer encoding is used. For the remaining ones, Inc-WMSU3 outperforms WMSU3 by a factor of 2. These results underscore the importance of incrementality for speeding up MaxSAT algorithms.



(a) Run times of LinearUS and Inc-LinearUS (b) Run times of WMSU3 and Inc-WMSU3

Figure 7: Impact of incremental encodings (• Iterative Totalizer, □ Iterative SWC)

6. Conclusions and Future Work

The most effective MaxSAT algorithms for industrial instances are usually based on solving a sequence of SAT formulas. More recently, there has been a surge of new unsatisfiability-based algorithms where all but the last SAT formula are unsatisfiable. As a result, in most cases, the SAT formula is completely rebuilt between iterations and knowledge acquired in the previous iterations is lost.

In this paper we propose to apply incrementality to unsatisfiability-based MaxSAT algorithms. This is achieved by incrementally encoding cardinality and pseudo-Boolean constraints, such that the SAT formula can be iteratively changed between iterations. As a result, only one instance of the SAT solver is created, thus maintaining the learned clauses and the internal state of the solver from one iteration to the next.

Experimental results on the MaxSAT Evaluation 2014 instances show that the incremental algorithms clearly outperform the non-incremental versions for both weighted and unweighted MaxSAT. Furthermore, the paper shows that by using incrementality, a very simple MaxSAT algorithm can be competitive with more sophisticated algorithms for several sets of instances. Considering the complementary results from the different solvers and configurations, a portfolio-based MaxSAT solver can be easily built on the open-wbo framework.

The proposed techniques are not specific to the WMSU3 algorithm. For example, these techniques can be used in other unsatisfiability-based algorithms such as WPM [3], BCD2 [48], and MSCG [46]. Moreover, the application of the proposed techniques are not confined to MaxSAT solving. For example, they have been used in other application domains such as automated fence insertion in programs under weak memory models [29]. As future work, we propose to integrate these techniques into more application domains where cardinality and pseudo-Boolean constraints need to be updated iteratively. Furthermore,

we also plan to extend incrementality to other effective cardinality and pseudo-Boolean constraints encodings.

Acknowledgments

This work is partially supported by the ERC project 280053, Fundação para a Ciência e a Tecnologia (FCT) grant POLARIS (PTDC/EIA-CCO/123051/2010), and national funds through FCT with reference UID/CEC/50021/2013.

References

[1] Ignasi Abío and Peter J. Stuckey. Conflict directed lazy decomposition. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, **7514** of *LNCS*, pages 70–85. Springer, 2012.

[2] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In Agostino Dovier and Vítor Santos Costa, editors, *International Conference on Logic Programming*, **17** of *LIPIcs*, pages 211–221. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[3] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving WPM2 for (weighted) partial maxsat. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, **8124** of *LNCS*, pages 117–132. Springer, 2013.

[4] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In Kullmann [32], pages 427–440.

[5] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial MaxSAT. In Maria Fox and David Poole, editors, *AAAI Conference on Artificial Intelligence*, pages 3–8. AAAI Press, 2010.

[6] Josep Argelich, Daniel Le Berre, Inês Lynce, João Marques-Silva, and Pascal Rapicault. Solving linux upgradeability problems using Boolean optimization. In Inês Lynce and Ralf Treinen, editors, *Workshop on Logics for Component Configuration*, **29** of *EPTCS*, pages 11–22, 2010.

[7] Roberto Asín and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals OR*, **218**(1):71–91, 2014.

[8] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, **16**(2):195–221, 2011.

[9] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In Matti Järvisalo and Allen Van Gelder, editors, *International Conference on Theory and Applications of Satisfiability Testing*, **7962** of *LNCS*, pages 309–317. Springer, 2013.

[10] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming*, **2833** of *LNCS*, pages 108–122. Springer, 2003.

[11] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In Kullmann [32], pages 181–194.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. www.SMT-LIB.org.

[13] Markus Büttner and Jussi Rintanen. Satisfiability planning with constraints on the number of actions. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *International Conference on Automated Planning and Scheduling*, pages 292–299. AAAI Press, 2005.

[14] Yibin Chen, Sean Safarpour, João Marques-Silva, and Andreas G. Veneris. Automated design debugging with maximum satisfiability. *IEEE Transactions on CAD of Integrated Circuits and Systems*, **29**(11):1804–1817, 2010.

[15] Alessandro Cimatti and Roberto Sebastiani, editors. *International Conference on Theory and Applications of Satisfiability Testing*, **7317** of *LNCS*. Springer, 2012.

[16] Romuald Debruyne. Arc-consistency in dynamic CSPs is no more prohibitive. In Mark G. Radle, editor, *International Conference on Tools with Artificial Intelligence*, pages 299–307. IEEE, 1996.

[17] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *AAAI Conference on Artificial Intelligence*, pages 37–42. AAAI Press / The MIT Press, 1988.

[18] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, **4144** of *LNCS*, pages 81–94. Springer, 2006.

[19] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *International Conference on Theory and Applications of Satisfiability Testing*, **2919** of *LNCS*, pages 502–518. Springer, 2003.

[20] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, **89**(4):543–560, 2003.

[21] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:1–26, 2006.

[22] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *International Conference on Theory and Applications of Satisfiability Testing*, **4121** of *LNCS*, pages 252–265. Springer, 2006.

[23] Ana Graça, Inês Lynce, João Marques-Silva, and Arlindo L. Oliveira. Efficient and accurate haplotype inference by combining parsimony and pedigree information. In Katsuhisa Horimoto, Masahiko Nakatsui, and Nikolaj Popov, editors, *Algebraic and Numeric Biology*, **6479** of *LNCS*, pages 38–56. Springer, 2010.

[24] Federico Heras, António Morgado, and João Marques-Silva. Core-guided binary search algorithms for maximum satisfiability. In Wolfram Burgard and Dan Roth, editors, *AAAI Conference on Artificial Intelligence*. AAAI Press, 2011.

[25] Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In Birte Glimm and Antonio Krüger, editors, *KI 2013: Advances in Artificial Intelligence*, **7526** of *LNCS*, pages 107–118. Springer, 2012.

[26] John N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, **15**(1&2):177–186, 1993.

[27] Alexey Ignatiev, António Morgado, Vasco Manquinho, Inês Lynce, and João Marques-Silva. Progression in maximum satisfiability. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *European Conference on Artificial Intelligence*, **263** of *Frontiers in Artificial Intelligence and Applications*, pages 453–458. IOS Press, 2014.

[28] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In Mary W. Hall and David A. Padua, editors, *Programming Language Design and Implementation*, pages 437–446. ACM, 2011.

[29] Saurabh Joshi and Daniel Kroening. Property-driven fence insertion using reorder bounded model checking. In Nikolaj Bjørner and Frank D. de Boer, editors, *Formal Methods*, **9109** of *LNCS*, pages 291–307. Springer, 2015.

[30] Serdar Kadioglu, Yuri Malitsky, and Meinolf Sellmann. Non-model-based search guidance for set partitioning problems. In Jörg Hoffmann and Bart Selman, editors, *AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.

[31] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT: a partial Max-SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, **8**(1/2):95–100, 2012.

[32] Oliver Kullmann, editor. *International Conference on Theory and Applications of Satisfiability Testing*, **5584** of *LNCS*. Springer, 2009.

[33] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, **7**(2-3):59–6, 2010.

[34] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[35] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal Automated Reasoning*, **40**(1):1–33, 2008.

[36] Florian Lonsing and Uwe Egly. Incremental QBF solving. In O'Sullivan [52], pages 514–530.

[37] Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: an efficient SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *International Conference on Theory and Applications of Satisfiability Testing*, **3542** of *LNCS*, pages 360–375. Springer, 2004.

[38] Panagiotis Manolios and Vasilis Papavasileiou. Pseudo-Boolean solving by incremental translation to SAT. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design*, pages 41–45. FMCAD Inc., 2011.

[39] Vasco Manquinho, João Marques-Silva, and Jordi Planes. Algorithms for weighted Boolean optimization. In Kullmann [32], pages 495–508.

[40] Paolo Marin, Christian Miller, Matthew D. T. Lewis, and Bernd Becker. Verification of partial designs using incremental QBF solving. In Wolfgang Rosenstiel and Lothar Thiele, editors, *Design, Automation, and Test in Europe Conference*, pages 623–628. IEEE, 2012.

[41] João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, **62**(3-4):317–343, 2011.

[42] João Marques-Silva and Jordi Planes. On using unsatisfiability for solving maximum satisfiability. *CoRR*, **abs/0712.1097**, 2007. http://arxiv.org/abs/0712.1097.

[43] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In O'Sullivan [52], pages 531–548.

[44] Ruben Martins, Vasco Manquinho, and Inês Lynce. Parallel search for maximum satisfiability. *AI Communications*, **25**(2):75–95, 2012.

[45] Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: a modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *International Conference on Theory and Applications of Satisfiability Testing*, **8561** of *LNCS*, pages 438–445. Springer, 2014.

[46] António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In O'Sullivan [52], pages 564–573.

[47] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: a survey and assessment. *Constraints*, **18**(4):478–534, 2013.

[48] António Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for MaxSAT. In Cimatti and Sebastiani [15], pages 284–297.

[49] Alexander Nadel and Vadim Ryvchin. Efficient sat solving under assumptions. In Cimatti and Sebastiani [15], pages 242–255.

[50] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In Carla E. Brodley and Peter Stone, editors, *AAAI Conference on Artificial Intelligence*, pages 2717–2723. AAAI Press, 2014.

[51] Toru Ogawa, Yangyang Liu, Ryuzo Hasegawa, Miyuki Koshimura, and Hiroshi Fujita. Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers. In Nikolaos Bourbakis, editor, *International Conference on Tools with Artificial Intelligence*, pages 9–17. IEEE, 2013.

[52] Barry O'Sullivan, editor. *Principles and Practice of Constraint Programming*, **8656** of *LNCS*. Springer, 2014.

[53] Sven Reimer, Matthias Sauer, Tobias Schubert, and Bernd Becker. Incremental encoding and solving of cardinality constraints. In Franck Cassez and Jean-Francois Raskin, editors, *Automated Technology for Verification and Analysis*, **8837** of *LNCS*, pages 297–313. Springer, 2014.

[54] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming*, **3709** of *LNCS*, pages 827–831. Springer, 2005.

[55] Ofer Strichman. Pruning techniques for the SAT-based bounded model checking problem. In Tiziana Margaria and Thomas F. Melham, editors, *Correct Hardware Design and Verification Methods*, **2144** of *LNCS*, pages 58–70. Springer, 2001.

[56] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, **68**(2):63–69, 1998.

[57] Jesse Whittemore, Joonyoung Kim, and Karem A. Sakallah. SATIRE: A new incremental satisfiability engine. In Jan Rabaey, editor, *Design Automation Conference*, pages 542–545. ACM, 2001.