Incremental Cardinality Constraints for MaxSAT*

Ruben Martins[1], Saurabh Joshi[1], Vasco Manquinho[2], and Inês Lynce[2]

[1]University of Oxford, Department of Computer Science, United Kingdom
{ruben.martins,saurabh.joshi}@cs.ox.ac.uk
[2]INESC-ID / Instituto Superior Técnico, Universidade de Lisboa, Portugal
{vmm,ines}@sat.inesc-id.pt

Abstract. Maximum Satisfiability (MaxSAT) is an optimization variant of the Boolean Satisfiability (SAT) problem. In general, MaxSAT algorithms perform a succession of SAT solver calls to reach an optimum solution making extensive use of cardinality constraints. Many of these algorithms are non-incremental in nature, i.e. at each iteration the formula is rebuilt and no knowledge is reused from one iteration to another. In this paper, we exploit the knowledge acquired across iterations using novel schemes to use cardinality constraints in an incremental fashion. We integrate these schemes with several MaxSAT algorithms. Our experimental results show a significant performance boost for these algorithms as compared to their non-incremental counterparts. These results suggest that incremental cardinality constraints could be beneficial for other constraint solving domains.

1 Introduction

Plethora of application domains such as software package upgrades [5], error localization in C code [27], debugging of hardware designs [12], haplotyping with pedigrees [24], and course timetabling [6] have benefited from the advancement in MaxSAT solving techniques. Considering such diversity of application domains for MaxSAT algorithms, the continuous improvement of MaxSAT solving techniques is imperative.

Incremental approaches have provided a huge leap in the performance of SAT solvers [47, 22, 45, 8]. However, the notion of incrementality has not yet been fully exploited in MaxSAT solving. Most MaxSAT algorithms perform a succession of SAT solver calls to reach optimality. Incremental approaches 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 [23, 38, 25, 43] 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.

* Partially supported by the ERC project 280053, and FCT grants ASPEN (PTDC/EIA-CCO/110921/2009), POLARIS (PTDC/EIA-CCO/123051/2010), and INESC-ID's multiannual PIDDAC funding PEst-OE/EEI/LA0021/2013.

Between the iterations of a MaxSAT algorithm, cardinality constraints are added to the formula [23, 3, 25, 43]. Usually, cardinality constraints are encoded in CNF so that a SAT solver can handle the resulting formula [9, 46, 7]. Otherwise, calls to a SAT solver must be replaced with calls to a pseudo-Boolean solver that natively handles cardinality constraints [38]. This paper discusses the use of cardinality constraints in an incremental manner to enhance MaxSAT algorithms. To achieve this, we propose the following incremental approaches: (i) incremental blocking, (ii) incremental weakening, and (iii) iterative encoding.

The remainder of the paper is organized as follows. Section 2 introduces preliminaries and notations. We describe our proposed techniques in Section 3. In Section 4, we mention prior research work done in relevant areas. We show the superiority of our approaches through experimental results in Section 5. Finally, Section 6 presents concluding remarks.

2 Preliminaries

A Boolean formula in conjunctive normal form (CNF) is a conjunction of clauses, where a clause is a disjunction of literals and 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 $true$ ($false$). A literal x_i ($\neg x_i$) is said 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 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 ω is a set of literals.

Maximum Satisfiability (MaxSAT) 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). From now on, 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 [33]. 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. The weighted version of MaxSAT allows soft clauses to have weights greater than or equal to 1 and the objective is to satisfy all hard clauses while minimizing the total weight of unsatisfied soft clauses. In this paper we assume a partial MaxSAT formula. The described algorithms can be generalized to the weighted versions of MaxSAT.

Cardinality constraints are a generalization of propositional clauses. In a cardinality constraint, a sum of n literals must be smaller than or equal to a given value k, i.e. $\sum_{i=1}^{n} l_i \leq k$ where l_i is a literal. As a result, a cardinality constraint over n literals ensures that at most k literals can be satisfied.

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** $\omega \in \varphi_s$ **do**
3 $\quad V_R \leftarrow V_R \cup \{r\}$ `// r is a new relaxation variable`
4 $\quad \omega_R \leftarrow \omega \cup \{r\}$
5 $\quad \varphi_W \leftarrow \varphi_W \cup \{\omega_R\}$
6 **while** true **do**
7 $\quad (\mathsf{st}, \nu, \varphi_C) \leftarrow \mathtt{SAT}(\varphi_W \cup \{\mathtt{CNF}(\sum_{r \in V_R} r \leq \lambda)\}, \emptyset)$
8 $\quad$ **if** $\mathsf{st} = \mathsf{SAT}$ **then**
9 $\quad\quad$ **return** ν `// satisfying assignment to` φ
10 $\quad \lambda \leftarrow \lambda + 1$

2.1 MaxSAT Algorithms

Due to the recent developments in SAT solving, different algorithms for solving MaxSAT have been proposed that rely on multiple calls to a SAT solver. A SAT solver call $\mathtt{SAT}(\varphi, \mathcal{A})$ receives 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. Assumptions may lead to early termination if the SAT solver learns a clause where at least one of the literals in $\mathcal{A}$ must be unsatisfied. An assumption controls the value of a variable for a given SAT call, whereas a unit clause controls the value of a variable for all the SAT calls after the unit clause has been added.

The SAT call returns 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 the reason of unsatisfiability. 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.

The algorithms presented in the paper assume that a SAT solver call is previously performed 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.

Algorithm 1 performs a linear search on the number of unsatisfied soft clauses. First, a new relaxation variable r is added to each soft clause ω (lines 2-5). The goal is to find an assignment to the input variables that minimizes the number of relaxation variables that are assigned value $true$. If the original clause ω is unsatisfied, then r is assigned to $true$. At each iteration, a cardinality constraint is defined such that at most λ relaxation variables can be assigned to $true$. This cardinality constraint is encoded into CNF and given to the SAT solver (line 7). Algorithm 1 starts with $\lambda = 0$ and in each iteration λ is increased until the SAT solver finds a satisfying assignment. Hence, λ defines a lower bound on the number of unsatisfied soft clauses of φ. At each iteration, the result of the SAT call is UNSAT, except the last one that provides an optimal solution to φ.

Algorithm 2: Fu-Malik Algorithm

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

1 $(\varphi_W, \varphi_{W_s}) \leftarrow (\varphi, \varphi_s)$
2 **while** true **do**
3 $\quad (\mathsf{st}, \nu, \varphi_C) \leftarrow \texttt{SAT}(\varphi_W, \emptyset)$
4 $\quad$ **if** $\mathsf{st} = \mathsf{SAT}$ **then**
5 $\quad\quad$ **return** ν `// satisfying assignment to` φ
6 $\quad V_R \leftarrow \emptyset$
7 $\quad$ **foreach** $\omega \in (\varphi_C \cap \varphi_{W_s})$ **do**
8 $\quad\quad V_R \leftarrow V_R \cup \{r\}$ `// r is a new relaxation variable`
9 $\quad\quad \omega_R \leftarrow \omega \cup \{r\}$
10 $\quad\quad \varphi_{W_s} \leftarrow (\varphi_{W_s} \setminus \{\omega\}) \cup \{\omega_R\}$
11 $\quad\quad \varphi_W \leftarrow (\varphi_W \setminus \{\omega\}) \cup \{\omega_R\}$
12 $\quad \varphi_W \leftarrow \varphi_W \cup \{\texttt{CNF}(\sum_{r \in V_R} r \leq 1)\}$

Algorithm 3: MSU3 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 $\quad (\mathsf{st}, \nu, \varphi_C) \leftarrow \texttt{SAT}(\varphi_W \cup \{\texttt{CNF}(\sum_{r \in V_R} r \leq \lambda)\}, \emptyset)$
4 $\quad$ **if** $\mathsf{st} = \mathsf{SAT}$ **then**
5 $\quad\quad$ **return** ν `// satisfying assignment to` φ
6 $\quad$ **foreach** $\omega \in (\varphi_C \cap \varphi_s)$ **do**
7 $\quad\quad V_R \leftarrow V_R \cup \{r\}$ `// r is a new variable`
8 $\quad\quad \omega_R \leftarrow \omega \cup \{r\}$ `//` ω `was not previously relaxed`
9 $\quad\quad \varphi_W \leftarrow (\varphi_W \setminus \{\omega\}) \cup \{\omega_R\}$
10 $\quad \lambda \leftarrow \lambda + 1$

Algorithm 1 follows an Unsat-Sat linear search. A converse approach is the Sat-Unsat linear search where λ is defined as an upper bound. In that case, λ is initialized with the number of soft clauses. Next, while the SAT call is satisfiable, λ is decreased. The algorithm ends when the SAT call returns UNSAT and the last satisfying assignment found is an optimal solution to φ.

Core-guided algorithms for MaxSAT take advantage of the certificates of unsatisfiability produced by the SAT solver [23]. In Algorithm 2, proposed by Fu and Malik [23], soft clauses are only relaxed when they appear in some unsatisfiable core φ_C returned by the SAT solver. Initially, we consider all hard and soft clauses without relaxation. In each iteration, an unsatisfiable subformula φ_C is identified and relaxed by adding a new relaxation variable to each soft clause in φ_C (lines 7-11). Additionally, a new constraint is added such that at most one of the new relaxation variables can be assigned to $true$ (line 12). The algorithm stops when the formula becomes satisfiable.

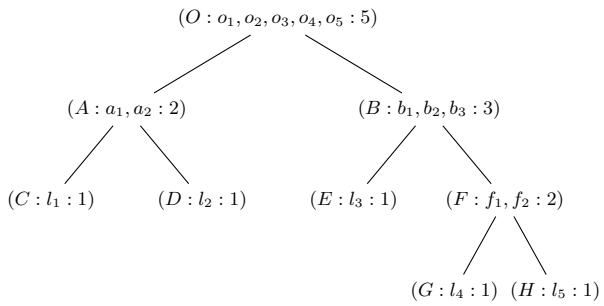


Fig. 1: Totalizer encoding for $l_1 + \cdots + l_5 \leq k$

In Algorithm 2 soft clauses may have to be relaxed several times. As a result, several relaxation variables can be added to the same soft clause. Nevertheless, other core-guided algorithms have already been proposed where at most one relaxation variable is added to each soft clause [3, 40]. Algorithm 3 follows a linear search Unsat-Sat, but soft clauses are only relaxed when they appear in some unsatisfiable core φ_C.

In this section we solely describe MaxSAT algorithms that will be the focus of the enhancements proposed in the paper. We refer to the literature for other approaches such as branch and bound algorithms using MaxSAT inference techniques or procedures to estimate the number of unsatisfied clauses to prune the search [33]. Additionally, there is also an extended overview on core-guided algorithms [43].

2.2 Totalizer Encoding

For the purpose of this paper, we describe the Totalizer encoding [9] for cardinality constraints, as later in the paper we build upon this encoding to present our novel approaches. Totalizer encoding can be better visualized as a tree as shown in Fig. 1. Here, 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 in unary. Therefore, at every node its corresponding $node\_vars$ represents an integer from 1 to $node\_sum$ in the 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 where as the root node has the output variables $(o_1, \ldots, o_5)$ giving the finally 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, Eq. 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 r_σ must be set to $true$ to indicate that at least $\alpha + \beta$ many leaves have been set to $true$ under P. Eq. 1 only counts the number of input literals set to $true$. In other words, it encodes *cardinality sum* over 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 from (l_3, l_4, l_5) where as, the tree rooted at A counts the set literals from (l_1, l_2).*

Note also that Eq. 1 counts up to n and then Eq. 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, Eq. 1 remains valid. However, the equality $n_2 + n_3 = n_1$ may no longer hold. With this modification, Eq. 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 $O(nk)$ [11, 29]. 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 Eq. 1 and k-simplification. Observe that $\varphi_1 \subset \varphi_2$, whenever $k_1 < k_2$.*

3 Incremental Approaches

MaxSAT algorithms that are based on refining unsatisfiable SAT formulas can be enhanced by changing cardinality constraints in an incremental fashion. In this section, we propose the following three techniques to enable incrementality when using cardinality constraints: (i) incremental blocking, (ii) incremental weakening, and (iii) iterative encoding.

3.1 Incremental Blocking

MaxSAT algorithms based on refining unsatisfiable formulas are usually non-incremental. After an unsatisfiable iteration, the formula is refined by removing a certain set of clauses and adding a new set of clauses that imposes a weaker constraint over the relaxation variables. However, SAT solvers do not allow the deletion of clauses that belong to the original formula. Since learned clauses from previous iterations may depend on the clauses that are now being removed, it is not sound to keep all of the learned clauses.

Incremental SAT solving addresses these problems by using assumptions [22]. To the best of our knowledge this approach has not been extended for incremental MaxSAT solving.

We denote b as a *blocking variable* which is used to extend a clause ω to $(\omega \vee b)$. When b is set to $false$ the original clause ω is enforced (enabled). When b is set to $true$ the extended clause $(\omega \vee b)$ is trivially satisfied and ω is no longer enforced (disabled). Thus, adding b (or $\neg b$) as an assumption or unit clause disables (or enables) a clause. Using a blocking variable, we can overcome the limitation of a SAT solver not allowing clause deletions.

MaxSAT Algorithms based on Cardinality Constraints. Many MaxSAT algorithms are based on refining the formula by encoding and updating cardinality constraints [25, 2, 43]. For these algorithms, the incremental blocking can be done when cardinality constraints are encoded to CNF.

$$\varphi \boxplus b \equiv \{\omega \vee b : \omega \in \varphi\} \tag{3a}$$

$$\Psi(\mathbf{X}, k, b) \equiv \texttt{CNF}_{Tot^k}(\Sigma x_i) \boxplus b \tag{3b}$$

$$\varphi^i \equiv \varphi_W \cup \left(\bigcup_{j=1}^{i} \Psi(\mathbf{X^j}, k^j, b^j) \right) \cup \langle \neg b^i, \neg o_{k^i+1} \rangle \cup \left[b^1, \ldots, b^{i-1} \right] \tag{3c}$$

$$\varphi^{i+1} \equiv \varphi_W \cup \left(\bigcup_{j=1}^{i+1} \Psi(\mathbf{X^j}, k^j, b^j) \right) \cup \langle \neg b^{i+1}, \neg o_{k^{i+1}+1} \rangle \cup \left[b^1, \ldots, b^{i} \right] \tag{3d}$$

Let Eq. 3a define the extension of a CNF formula φ with a blocking variable b. Next, $\Psi(\mathbf{X}, k, b)$ represents a cardinality sum up to $k+1$ over $x_1, \ldots, x_n$ encoded in CNF using Eq. 1 and k-simplification of the Totalizer encoding and extended with a blocking variable b. Then, for incremental blocking, at line 7 in Algorithm 1 and line 3 in Algorithm 3 we call the solver on φ^i as defined in Eq. 3c for the i^{th} iteration. Assumption $\langle \neg b^i \rangle$ enables the cardinality constraint for the current iteration whereas unit clauses $\left[b^1, \ldots, b^{i-1} \right]$ ensure that cardinality constraints from earlier iterations are disabled. In addition, assumption $\langle \neg o_{k^i+1} \rangle$ restricts the sum to k^i. Notice that in the $(i+1)^{th}$ iteration, a new cardinality sum $\Psi(\mathbf{X}^{i+1}, k^{i+1}, b^{i+1})$ is added and earlier constraints are disabled as assumption $\langle \neg b^i \rangle$ moves as unit clause $\left[b^i \right]$.

Assume the MaxSAT formula has a given optimum value k_{opt}. When considering Algorithm 1 and the Totalizer encoding, incremental blocking creates an encoding for each k^i up to k_{opt}. Hence, the overall encoding would have $O(\sum_{i=0}^{k_{opt}} ni) = O(nk_{opt}^2)$ auxiliary clauses. Though incremental blocking creates more clauses as compared to a non-incremental approach ($O(nk_{opt})$), keeping the inner state of the constraint solver across iterations significantly reduces the solving time. A similar reasoning can be made for Algorithm 3 or any other MaxSAT algorithm that uses incremental blocking.

Fu-Malik Algorithm with Incremental Blocking. Incremental blocking can also be used for MaxSAT algorithms that do not update cardinality constraints but modify the

Algorithm 4: Fu-Malik Algorithm with Incremental Blocking

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

1 $(\varphi_W, \varphi_{W_s}, \mathcal{A}, \mathcal{B}) \leftarrow (\varphi, \varphi_s, \emptyset, \emptyset)$
2 **while** true **do**
3 $\quad (\mathsf{st}, \nu, \varphi_C) \leftarrow \texttt{SAT}(\varphi, \mathcal{A})$
4 $\quad$ **if** $\mathsf{st} = \mathsf{SAT}$ **then**
5 $\quad\quad$ **return** ν `// satisfying assignment to` φ
6 $\quad V_R \leftarrow \emptyset$
7 $\quad$ **foreach** $\omega \in (\varphi_C \cap \varphi_{W_s})$ **do**
8 $\quad\quad V_R \leftarrow V_R \cup \{r\}$ `// r is a new relaxation variable`
9 $\quad\quad \omega_R \leftarrow (\omega \setminus \mathcal{B}) \cup \{r\} \cup \{b\}$ `// b is a new blocking variable`
10 $\quad\quad \mathcal{B} \leftarrow \mathcal{B} \cup \{b\}$
11 $\quad\quad \varphi_{W_s} \leftarrow (\varphi_{W_s} \setminus \{\omega\}) \cup \{\omega_R\}$
12 $\quad\quad \mathcal{A} \leftarrow (\mathcal{A} \setminus \{\neg b' : b' \in \mathcal{B} \cap \omega\}) \cup \{\neg b\}$ `// enables` ω_R
13 $\quad\quad \varphi_W \leftarrow \varphi_W \cup \{\omega_R\} \cup \{b' : b' \in \mathcal{B} \cap \omega\}$ `// disables` ω
14 $\quad \varphi_W \leftarrow \varphi_W \cup \{\texttt{CNF}(\sum_{r \in V_R} r \leq 1)\}$

formula at each iteration. For example, Fu-Malik algorithm (Algorithm 2, Section 2) can be enhanced with incremental blocking. Algorithm 4 shows the modifications to Fu-Malik algorithm to support incremental blocking. The main differences between the incremental and non-incremental versions of Fu-Malik algorithm are highlighted. For each soft clause ω in φ_C, Algorithm 4 copies ω into ω_R without blocking variables (line 9). Next, it adds a fresh blocking variable b and a fresh relaxation variable r to ω_R (line 9). The current soft clause ω_R is enabled by adding $\langle \neg b \rangle$ to the set of assumptions, where b is the blocking variable that occurs in ω_R (line 12). At the same time, the assumption $\langle \neg b' \rangle$ is removed from the set of assumptions, where b' is the blocking variable that occurs in ω (line 12). Finally, the working formula φ_W is updated with the new clause ω_R, and with the unit clause $[b']$. Note that this unit clause disables ω from the working formula φ_W since ω contains b' and therefore is always satisfied.

The incremental version of Fu-Malik algorithm creates m auxiliary clauses at each iteration, where m is the number of soft clauses in the unsatisfiable subformula. However, the size of unsatisfiable subformulas tends to be small when compared to the total number of soft clauses. Note that the number of auxiliary clauses created by the incremental version of Fu-Malik is not as large as when incremental blocking is directly applied to cardinality encodings.

3.2 Incremental Weakening

Since incremental blocking encodes a new cardinality constraint at each iteration, this results in an increase in formula size at every iteration. To circumvent this increase, one can build the cardinality sum only once, and incrementally weaken the cardinality bound (k).

Incremental weakening is similar to *incremental strengthening* [7], but instead of constraining the output of the cardinality constraint with unit clauses it uses assumptions. Notice that incremental strengthening is used in linear search Sat-Unsat algorithms. In these algorithms, the cardinality bound decreases monotonically at each iteration. Therefore, the unit clauses that constrain the previous cardinality bound remain valid when considering the new bound. On the other hand, incremental weakening is used for MaxSAT algorithms that search on the lower bound of the optimal solution. For these algorithms, the restriction of the cardinality bound is only valid for the current iteration and must be updated for the upcoming iterations.

$$\Gamma(\mathbf{X}, k) \equiv \texttt{CNF}_{Tot^k}(\Sigma x_i) \tag{4a}$$

$$\varphi^i \equiv \varphi_W \cup \Gamma(\mathbf{X}, k_u) \cup \langle \neg o_{k^i+1}, \ldots, \neg o_{k_u} \rangle \tag{4b}$$

$$\varphi^{i+1} \equiv \varphi_W \cup \Gamma(\mathbf{X}, k_u) \cup \langle \neg o_{k^{i+1}+1}, \ldots, \neg o_{k_u} \rangle \tag{4c}$$

Let $\Gamma(\mathbf{X}, k)$ be the cardinality sum over input literals $x_1, \ldots, x_n$ encoded in CNF using Eq. 1 and k-simplification. Then, for incremental weakening, at line 7 in Algorithm 1 and line 3 in Algorithm 3 we call the solver on φ^i as defined in Eq. 4b for the i^{th} iteration. Note that $\Gamma(\mathbf{X}, k_u)$ is encoded only once for a conservative upper bound k_u. For the i^{th} iteration, we restrict the cardinality sum to k^i using assumptions $\langle \neg o_{k^i+1}, \ldots, \neg o_{k_u} \rangle$ (Eq. 2). In the following iteration (Eq. 4c), we only change assumptions to restrict the cardinality sum to k^{i+1}.

To obtain a conservative upper bound k_u, we invoke the SAT solver over φ_h to check if the set of hard clauses itself is satisfiable. If it is not satisfiable, the original MaxSAT formula φ can not be solved. However, if φ_h is satisfiable, one can count the number of soft clauses that remain unsatisfied under the satisfying assignment for φ_h. This number can be used as k_u since we know at least one assignment where k_u many clauses remain unsatisfied. Therefore, the optimum value k_{opt} must be smaller or equal to k_u.

With an upper bound k_u, incremental weakening creates $O(nk_u)$ auxiliary clauses as opposed to $O(nk_{opt})$ of the non-incremental approach. However, a non-incremental approach builds a new formula of size $O(nk_{opt})$ for every iteration, whereas incremental weakening builds the formula only once keeping the internal state and learned clauses across iterations. This results in a significant performance boost for MaxSAT algorithms using incremental weakening.

Incremental weakening does not allow the number of input literals in the cardinality constraint to change. Therefore, it does not directly support the MSU3 Algorithm (Algorithm 3, Section 2). To use incremental weakening with Algorithm 3, we modify the algorithm to relax all soft clauses and build a cardinality constraint over all relaxation variables. The relaxation variables r_i that do not appear in an unsatisfiable subformula φ_C are added as assumptions of the form $\langle \neg r_i \rangle$. This enforces the soft clauses corresponding to the relaxation variables until these clauses occur in φ_C. When they do occur, assumptions $\neg r_i$ are removed and their value is now only restricted by the cardinality constraint. Even though this procedure allows the incremental weakening approach to be used with Algorithm 3, it does not benefit from smaller encodings resulting from having less input literals in the cardinality constraint. Therefore, the

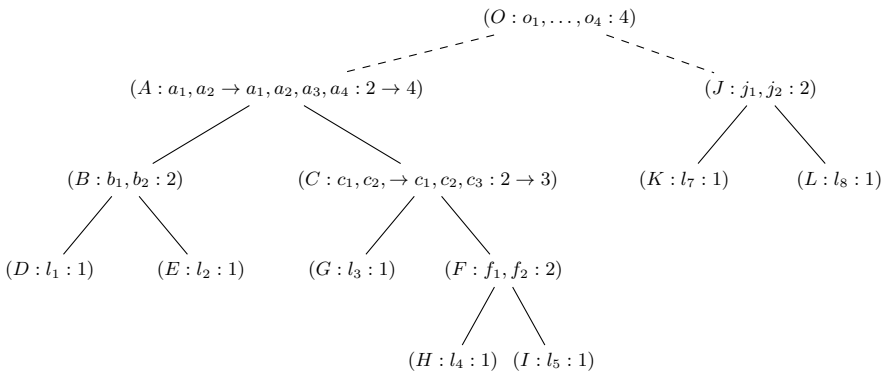


Fig. 2: Transforming $l_1 + \cdots + l_5 \leq 1$ and $l_7 + l_8 \leq 1$ into $l_1 + \ldots + l_5 + l_7 + l_8 \leq 3$

non-incremental approach may create a much smaller encoding than the incremental weakening approach for Algorithm 3.

3.3 Iterative Encoding

Incremental weakening uses a conservative upper bound (e.g., k_u) on the number of unsatisfied soft clauses in order to encode the cardinality constraint only once. However, this upper bound may be much larger than the optimum value (e.g. k_{opt}) which may result in a larger encoding than the non-incremental approach. In addition, incremental weakening does not allow the set of input literals in the cardinality constraint to change. Therefore, MaxSAT algorithms that increase the input literals of the cardinality constraint can not take 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 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 2.2. 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 Eq. 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 : (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 Eq. 1 and k-simplification in order to encode $x_1 + \cdots + x_n + y_1 + \cdots + y_m \leq k_2$. Since 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.

Linear search Unsat-Sat algorithm (Algorithm 1, Section 2) increases the cardinality 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 perform steps (1) and (2). On the other hand, MSU3 algorithm (Algorithm 3, Section 2) 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, 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. Iterative encoding is not only faster but allows us to solve more problem instances as compared to non-incremental approaches.

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 [50, 21] and SAT extensions [29, 8].

Assumptions are widely used for incremental SAT [22, 45]. The minisat solver [21] interface allows the definition of a set of assumptions. Alternatively, the interface of zchaff [36] allows removing groups of clauses.

Although not implemented, the work of Fu and Malik in MaxSAT [23] discusses how learned clauses may be kept from one SAT iteration to the next one. In Pseudo Boolean Optimization (PBO), early implementations include the use of incremental strengthening in minisat+ [20]. Linear search Sat-Unsat algorithms [29, 32] are implemented incrementally. A critical issue is on keeping *safe* learned clauses in successive iterations of a core-guided algorithm [41]. Quantified Boolean Formula (QBF) solving has successfully been made incremental [35] and further applied to verification [39].

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 [37]. 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.* [34]. Later on, the SAT solver Glucose has been made incremental using assumptions and applied to MUS extraction [8].

Incrementality is also present in other SAT-related domains such as Satisfiability Modulo Theories (SMT) and Bounded Model Checking (BMC). The SMT-LIB

v2.0 [10] defines the operations *push* and *pop* to work with a stack containing a set of formulas to be jointly solved. The MaxSAT solvers WPM1 and WPM2 [2] use the SMT solver Yices [19] which supports incrementality. Its use resembles the blocking strategy. The use of SAT solvers in BMC is known to benefit from incrementality, either by implementing incremental SAT solving [47] or by using assumptions [22].

In the context of Constraint Satisfaction Problems (CSPs), incremental formulations, incremental propagation and incremental solving are worth mentioning. Incrementality is naturally present in Dynamic CSPs (DCSPs) [18]. In DCSPs, 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 [17]. Incremental propagation in CSP [31, 13] makes use of *advisors* which give propagators a detailed view of the dynamic changes between propagator runs. Advisors enable the implementation of optimal algorithms for important constraints. Search in CSP is inherently incremental. From the first implementations, the approach to solve many CSPs is to incrementally build a solution, backtracking when an infeasibility is detected, until a solution is found or the problem is proven to have no solution [48]. More recently, incrementality has been implemented in global constraints mostly due to efficiency reasons [49].

5 Experimental Results

We used all partial MaxSAT instances (627) from the industrial category of the MaxSAT Evaluation 2013[1] as a benchmark for our experiments. The evaluation was performed on two AMD Opteron 6276 processors (2.3 GHz) running Fedora 18 with a timeout of 1,800 seconds and a memory limit of 8 GB. We implemented all algorithms described in section 2 (Linear search Unsat-Sat, Fu-Malik, and MSU3), as well as their incremental counterparts on top of OPEN-WBO [42]. OPEN-WBO is a modular open source MaxSAT solver that is easy to modify and is competitive with state-of-the-art MaxSAT solvers.

Table 1 shows the number of instances solved (*#Inst*) by the described MaxSAT algorithms using the different approaches, namely, non-incremental approach (*none*), incremental blocking (*blocking*), incremental weakening (*weakening*), and iterative encoding (*iterative*). Table 1 also shows the median speedup[2] for instances that have been solved by all incremental approaches for a given algorithm.

Fu-Malik with incremental blocking significantly outperforms the non-incremental algorithm. Incremental blocking not only solves more instances but also is significantly faster than the non-incremental algorithm. From those instances which were solved by both approaches, 50% of them have a speedup of at least 2.4. Incremental weakening and iterative encoding cannot be used with the Fu-Malik algorithm since it only uses at most one constraints and modifies the formula across iterations of the algorithm.

Linear search Unsat-Sat (LinearUS) with incremental blocking solves less instances than the non-incremental approach. Incremental blocking encodes a new cardinality constraint at each iteration of the MaxSAT algorithm, causing the formula to grow too

[1] Benchmarks available at `http://maxsat.ia.udl.cat/13/benchmarks/`

[2] The speedup of an instance is measured as the ratio of the solving time of the non-incremental approach to the solving time of the respective incremental approach.

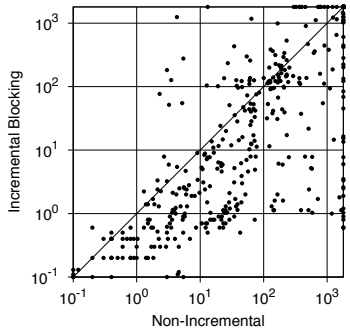
Table 1: Number of instances solved by the different incremental approaches and median speedup of solved instances

	None		Blocking		Weakening		Iterative	
	#Inst	Speedup	#Inst	Speedup	#Inst	Speedup	#Inst	Speedup
Fu-Malik	366	1.0	**388**	**2.4**	-	-	-	-
LinearUS	477	1.0	446	1.6	**498**	**2.3**	**509**	**2.4**
MSU3	517	1.0	488	1.6	504	2.0	**541**	**3.6**

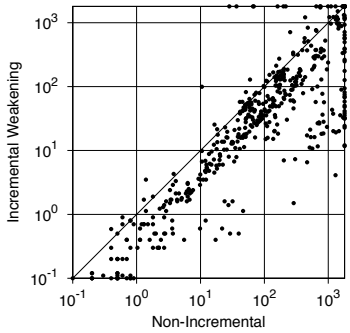
large resulting in termination due to memory outs. However, for those instances that were solved successfully, incremental blocking was 60% faster than the original LinearUS. Incremental weakening allows MaxSAT algorithms to solve more instances with significant speedup. Since the cardinality constraint is encoded only once, the size of the formula remains almost constant across iterations. The majority of the instances are solved at least $2\times$ faster. Iterative encoding outperforms all other approaches. Smaller formula sizes due to iterative encoding allows it to solve more instances as compared to incremental weakening.

MSU3 with incremental blocking solves less instances as compared to the original MSU3 but it is faster for instances solved by both approaches. Similar results have been observed for the LinearUS algorithm with incremental blocking. Incremental weakening outperforms incremental blocking in the number of solved instances as well as in terms of solving time. However, incremental weakening solves less instances than the non-incremental approach, since incremental weakening is not flexible to directly support the increase in the number of input literals of the cardinality constraint. A non-incremental approach may need to impose the cardinality constraint over a small subset of relaxation variables. Incremental weakening does not enjoy this benefit due to its inflexibility. This may result in incremental weakening producing a larger encoding for certain problem instances. Iterative encoding solves more instances and is significantly faster than the non-incremental approach. Iterative encoding only encodes the clauses that are needed at each iteration of the MaxSAT algorithm, allowing for an encoding with a similar size to the non-incremental approach. Most instances are solved at least $3.6\times$ faster with iterative encoding than without it.

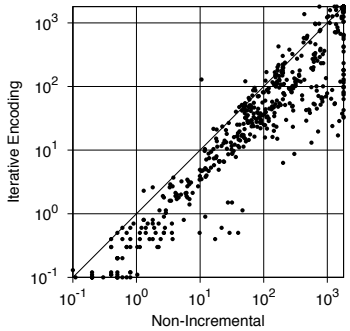
Fig. 3 shows scatter plots that compare the non-incremental and incremental approaches which are highlighted in Table 1. Each point in the plot corresponds to a problem instance, where the x-axis corresponds to the run time required by non-incremental approaches and the y-axis corresponds to the run time required by incremental approaches. Instances that are above the diagonal are solved faster when using a non-incremental approach, whereas instances that are below the diagonal are solved faster when using an incremental approach. Incremental approaches that we propose in this paper clearly assert their dominance over their non-incremental counterparts integrated with all three algorithms as shown in Fig. 3. This is particularly evident in the MSU3 algorithm where the majority of the instances are solved much faster with iterative encoding. For example, for 30% of the instances solved by MSU3 with and without



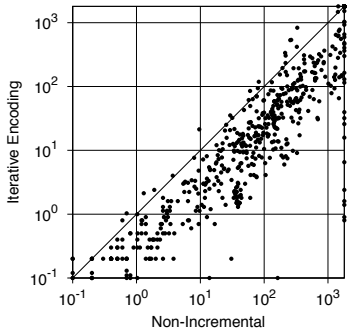
(a) Fu-Malik Algorithm:
Non-Incremental vs. Incremental Blocking



(b) LinearUS Algorithm:
Non-Incremental vs. Incremental Weakening



(c) LinearUS Algorithm:
Non-Incremental vs. Iterative Encoding



(d) MSU3 Algorithm:
Non-Incremental vs. Iterative Encoding

Fig. 3: Impact of incremental approaches

iterative encoding, iterative encoding is at least 6× faster than the non-incremental approach. For 10% of the instances solved by both approaches, iterative encoding boosts MSU3 with at least 14× speedup.

Fig. 4 shows a cactus plot with the running times of state-of-the-art MaxSAT solvers used in the MaxSAT Evaluation 2013[3] (WPM1 [3], WPM2 [4, 2], MaxHS [15, 16], BCD2 [44], QMaxSAT2 [29]) and the best incremental algorithms presented in this paper (incremental blocking Fu-Malik, iterative encoding LinearUS and MSU3).

Fu-Malik and WPM1 use similar MaxSAT algorithms. Moreover, WPM1 has a similar incremental strategy due to the incremental SMT solver that is used by WPM1. Since both solvers used similar techniques, it is not surprising that their performance is similar. Even though LinearUS uses a simple MaxSAT algorithm, it is competitive with more complex state-of-the-art MaxSAT algorithms. This is mostly due to the incremen-

[3] Only single engine solvers have been considered in this evaluation, therefore we did not include ISAC+ (a portfolio MaxSAT solver) [28].

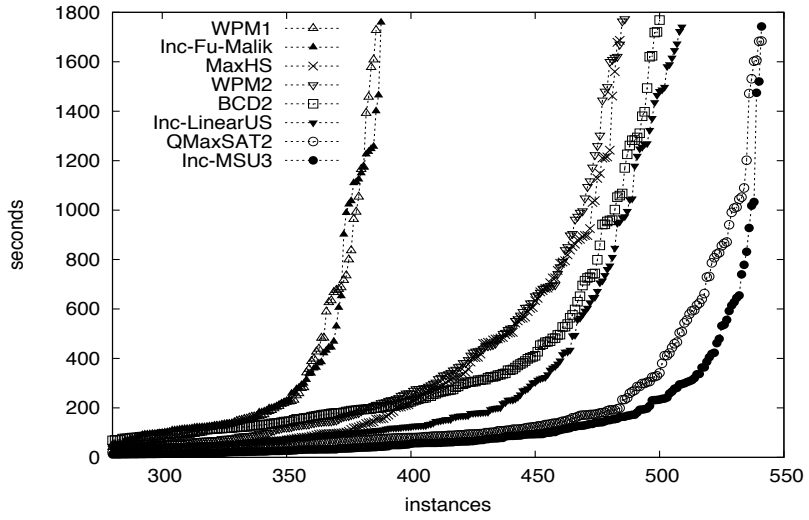


Fig. 4: Running times of state-of-the-art MaxSAT solvers

tal approach that is being used in LinearUS and shows the importance of using an efficient incremental approach. MSU3 and QMaxSAT2 perform complementary searches but both use incrementality and have similar performances. Iterative encoding is not restricted to MSU3 and may be used in other MaxSAT algorithms, such as WPM2 and BCD2. It is expected that if those algorithms are enhanced with the incremental iterative encoding, their performance might rise to values similar or higher than those of QMaxSAT2 and MSU3.

6 Conclusions and Future Work

Several state of the art MaxSAT algorithms are based on solving a sequence of closely related SAT formulas. However, although incrementality is not a new technique, it is seldom used in MaxSAT algorithms that search on the lower bound of the optimum solution. In this paper, we describe and propose new techniques to incrementally modify cardinality constraints used in several MaxSAT algorithms, namely in linear Unsat-Sat search, the classic Fu-Malik algorithm and MSU3 core-guided algorithm.

Experimental results show the effectiveness of the techniques proposed in the paper. The incremental versions of the MaxSAT algorithms clearly outperform the non-incremental versions, both in terms of speed and number of solved instances. Furthermore, the proposed techniques can be integrated in other core-guided algorithms such as WPM2 and BCD2, among others.

Finally, the paper also describes that in general it is possible to perform iterative encoding of cardinality constraints using the Totalizer encoding. Therefore, the use of this technique is not limited to the scope of MaxSAT algorithms. As future work, we propose to integrate these techniques in other domains where cardinality constraints are used, and to extend incrementality to other effective cardinality constraints encodings.

References

1. Abío, I., Stuckey, P.J.: Conflict Directed Lazy Decomposition. In: Milano, M. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 7514, pp. 70–85. Springer (2012)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving WPM2 for (Weighted) Partial MaxSAT. In: Principles and Practice of Constraint Programming. LNCS, vol. 8124, pp. 117–132. Springer (2013)
3. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In: Kullmann [30], pp. 427–440
4. Ansótegui, C., Bonet, M.L., Levy, J.: A New Algorithm for Weighted Partial MaxSAT. In: Fox, M., Poole, D. (eds.) AAAI Conference on Artificial Intelligence. AAAI Press (2010)
5. Argelich, J., Berre, D.L., Lynce, I., Marques-Silva, J., Rapicault, P.: Solving Linux Upgradeability Problems Using Boolean Optimization. In: Workshop on Logics for Component Configuration. pp. 11–22 (2010)
6. Asín, R., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. Annals of Operations Research pp. 1–21 (2012)
7. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality Networks: a theoretical and empirical study. Constraints 16(2), 195–221 (2011)
8. Audemard, G., Lagniez, J.M., Simon, L.: Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In: Järvisalo, M., Gelder, A.V. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 7962, pp. 309–317. Springer (2013)
9. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 2833, pp. 108–122. Springer (2003)
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010), available at `www.SMT-LIB.org`
11. Büttner, M., Rintanen, J.: Satisfiability Planning with Constraints on the Number of Actions. In: Biundo, S., Myers, K.L., Rajan, K. (eds.) International Conference on Automated Planning and Scheduling. pp. 292–299. AAAI (2005)
12. Chen, Y., Safarpour, S., Marques-Silva, J., Veneris, A.G.: Automated Design Debugging With Maximum Satisfiability. IEEE Transactions on CAD of Integrated Circuits and Systems 29(11), 1804–1817 (2010)
13. Cheng, K.C.K., Yap, R.H.C.: Maintaining Generalized Arc Consistency on Ad-Hoc n-Ary Boolean Constraints. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 78–82. IOS Press (2006)
14. Cimatti, A., Sebastiani, R. (eds.): Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, LNCS, vol. 7317. Springer (2012)
15. Davies, J., Bacchus, F.: Exploiting the Power of mip Solvers in maxsat. In: Järvisalo, M., Gelder, A.V. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 7962, pp. 166–181. Springer (2013)
16. Davies, J., Bacchus, F.: Postponing Optimization to Speed Up MAXSAT Solving. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 8124, pp. 247–262. Springer (2013)
17. Debruyne, R.: Arc-Consistency in Dynamic CSPs Is No More Prohibitive. In: International Conference on Tools with Artificial Intelligence. pp. 299–307. IEEE (1996)

18. Dechter, R., Dechter, A.: Belief Maintenance in Dynamic Constraint Networks. In: Shrobe, H.E., Mitchell, T.M., Smith, R.G. (eds.) AAAI Conference on Artificial Intelligence. pp. 37–42. AAAI Press / The MIT Press (1988)
19. Dutertre, B., de Moura, L.M.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. LNCS, vol. 4144, pp. 81–94. Springer (2006)
20. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)
21. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 2919, pp. 502–518. Springer (2003)
22. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
23. Fu, Z., Malik, S.: On Solving the Partial MAX-SAT Problem. In: Biere, A., Gomes, C.P. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 4121, pp. 252–265. Springer (2006)
24. Graça, A., Lynce, I., Marques-Silva, J., Oliveira, A.L.: Efficient and Accurate Haplotype Inference by Combining Parsimony and Pedigree Information. In: Algebraic and Numeric Biology. pp. 38–56. Springer (2010)
25. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Burgard, W., Roth, D. (eds.) AAAI Conference on Artificial Intelligence. AAAI Press (2011)
26. Hooker, J.N.: Solving the incremental satisfiability problem. Journal of Logic Programming 15(1&2), 177–186 (1993)
27. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Hall, M.W., Padua, D.A. (eds.) Programming Language Design and Implementation. pp. 437–446. ACM (2011)
28. Kadioglu, S., Malitsky, Y., Sellmann, M.: Non-Model-Based Search Guidance for Set Partitioning Problems. In: Hoffmann, J., Selman, B. (eds.) AAAI Conference on Artificial Intelligence. AAAI Press (2012)
29. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: A Partial Max-SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation 8, 95–100 (2012)
30. Kullmann, O. (ed.): Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, LNCS, vol. 5584. Springer (2009)
31. Lagerkvist, M.Z., Schulte, C.: Advisors for Incremental Propagation. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 409–422. Springer (2007)
32. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation 7(2-3), 59–6 (2010)
33. Li, C.M., Manyà, F.: MaxSAT, Hard and Soft Constraints. In: Handbook of Satisfiability, pp. 613–631. IOS Press (2009)
34. Liffiton, M.H., Sakallah, K.A.: Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. Journal Automated Reasoning 40(1), 1–33 (2008)
35. Lonsing, F., Egly, U.: Incremental QBF Solving. Computing Research Repository - arXiv abs/1402.2410 (2014)
36. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: An efficient sat solver. In: Hoos, H.H., Mitchell, D.G. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 3542, pp. 360–375. Springer (2004)

37. Manolios, P., Papavasileiou, V.: Pseudo-Boolean Solving by incremental translation to SAT. In: Bjesse, P., Slobodová, A. (eds.) International Conference on Formal Methods in Computer-Aided Design. pp. 41–45. FMCAD Inc. (2011)
38. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for Weighted Boolean Optimization. In: Kullmann [30], pp. 495–508
39. Marin, P., Miller, C., Lewis, M.D.T., Becker, B.: Verification of partial designs using incremental QBF solving. In: Rosenstiel, W., Thiele, L. (eds.) Design, Automation, and Test in Europe Conference. pp. 623–628. IEEE (2012)
40. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving Maximum Satisfiability. Tech. rep., Computing Research Repository, abs/0712.0097 (2007)
41. Martins, R., Manquinho, V., Lynce, I.: Parallel Search for Maximum Satisfiability. AI Communications 25(2), 75–95 (2012)
42. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: a Modular MaxSAT Solver. In: Sinz, C., Egly, U. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 8561, pp. 438–445. Springer (2014)
43. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: A survey and assessment. Constraints 18(4), 478–534 (2013)
44. Morgado, A., Heras, F., Marques-Silva, J.: Improvements to Core-Guided Binary Search for MaxSAT. In: Cimatti and Sebastiani [14], pp. 284–297
45. Nadel, A., Ryvchin, V.: Efficient SAT Solving under Assumptions. In: Cimatti and Sebastiani [14], pp. 242–255
46. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 3709, pp. 827–831. Springer (2005)
47. Strichman, O.: Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In: Margaria, T., Melham, T.F. (eds.) Correct Hardware Design and Verification Methods. LNCS, vol. 2144, pp. 58–70. Springer (2001)
48. van Beek, P.: Backtracking Search Algorithms . In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 4. Elsevier (2006)
49. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 6. Elsevier (2006)
50. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A New Incremental Satisfiability Engine. In: Design Automation Conference. pp. 542–545. ACM (2001)