# On Incremental Core-Guided MaxSAT Solving

Xujie Si[1], Xin Zhang[1], Vasco Manquinho[2], Mikoláš Janota[3], Alexey Ignatiev[4,5], and Mayur Naik[1]

[1] Georgia Institute of Technology, USA
[2] INESC-ID, IST, Universidade de Lisboa, Portugal
[3] Microsoft Research, Cambridge, UK
[4] LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
[5] ISDCT SB RAS, Irkutsk, Russia

**Abstract.** This paper aims to improve the efficiency of unsat core-guided MaxSAT solving on a sequence of similar problem instances. In particular, we consider the case when the sequence is constructed by adding new hard or soft clauses. Our approach is akin to the well-known idea of incremental SAT solving. However, we show that there are important differences between incremental SAT and incremental MaxSAT, where a straightforward implementation may lead to a sharp decrease in performance. We present alternatives that enable to cope with such issues. The presented algorithm is implemented and evaluated on practical problems. It solves more instances and yields an average speedup of 1.8× on previously solvable instances.

## 1 Introduction

MaxSAT is an optimization variant of the Boolean Satisfiability (SAT) problem. Recent years have witnessed vast improvements in the performance of MaxSAT solvers [1, 4–6, 14, 15, 24–26]. Emerging applications in a variety of domains pose large MaxSAT instances comprising tens of millions of clauses to such solvers.

A special but common scenario concerns applications which pose a *sequence* of *similar* large MaxSAT instances. For example, many applications involve a sequence of small updates to a large instance (e.g., verification via abstraction refinement [13,28] or user interaction [18]). Alternatively, MaxSAT-based solvers pose such sequences in order to scale to ever larger instances (e.g., using lazy [16] or demand-driven [29] methods) or more expressive theories (e.g., MaxSMT [7] and Markov Logic Networks [17, 27]). Instead of solving each instance in the sequence from scratch, it is desirable to improve the efficiency of MaxSAT solvers by reusing results computed across invocations on such instances.

In this paper, we focus on an especially common case in which the sequence of MaxSAT instances is constructed by *adding* hard or soft clauses. Moreover, the new clauses are determined by the solution to the previous instance. We target an unsat core-guided algorithm [12] which forms the basis of many popular MaxSAT solvers. This algorithm solves a single MaxSAT instance by solving a sequence of SAT instances until the underlying SAT solver finds a satisfying solution.

Each SAT instance is constructed using the unsat cores discovered in previous SAT instances. Since adding clauses to the current MaxSAT instance does not invalidate existing unsat cores, a compelling idea to improve the performance of solving the resulting MaxSAT instance is to reuse the existing unsat cores.

Surprisingly, however, we observe that a naive implementation of this idea can fail to yield performance benefits or, even worse, sharply curtain them. This reflects an inherent challenge to making core-guided MaxSAT solving incremental[6]: for a MaxSAT instance formed with two disjoint sets of clauses $\phi$ and $\delta$, solving $\phi$ followed by $\phi \cup \delta$, rather than solving $\phi \cup \delta$ directly, restricts the set of possible computations. This is because the set of unsat cores of $\phi$ is always a subset of those of $\phi \cup \delta$. Reusing the unsat cores of $\phi$ in solving $\phi \cup \delta$ can be detrimental because the MaxSAT algorithm's performance crucially depends on the quality of the unsat cores, and the unsat cores learnt from solving $\phi$ may be of poorer quality than those it would learn from solving $\phi \cup \delta$ directly.

To address this challenge, we propose a hybrid solving framework that alternates between the incremental algorithm and its non-incremental version. In each iteration, our framework checks whether the current instance may potentially benefit from reusing the cores learnt on previous instances. If the check succeeds, it applies the incremental algorithm by reusing such cores. Otherwise, it discards the cores learnt thus far and applies the non-incremental algorithm.

We implemented our approach in the Open-WBO MaxSAT solver [22] and evaluated it on 74 sequences generated from diverse applications in verification and information retrieval. Together, these sequences contain 669 MaxSAT instances, with an average of 10 million clauses per instance. Our evaluation shows that our approach outperforms the baseline approaches significantly: it yields an average speedup of 1.8× per sequence over the non-incremental approach, and it solves 19 more sequences than the naively-incremental approach.

## 2 Preliminaries

A propositional formula in *Conjunctive Normal Form (CNF)* is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal is either a Boolean variable $x_i$ or its negation $\neg x_i$. A literal $x_i$ ($\neg x_i$) is valued to true if $x_i$ is assigned to true (false). A literal $x_i$ ($\neg x_i$) is valued to false if $x_i$ is assigned to false (true). A clause is said to be *satisfied* if at least one of its literals is valued to true. If all literals in a clause are valued to false, the clause is said to be *unsatisfied*. We refer to CNF formulas as sets of clauses and clauses as sets of literals. For a CNF $\phi$, the *Satisfiability (SAT)* problem is defined as finding an assignment to all variables in $\phi$ that satisfies all clauses or determining that such an assignment does not exist.

The *Maximum Satisfiability (MaxSAT)* problem is an optimization version of SAT. Given a CNF formula $\phi$, the goal is to find a total assignment that min-

[6] Some works (e.g., [20]) define "incremental MaxSAT solving" as solving a MaxSAT instance by using a SAT solver incrementally. In this paper, it denotes solving a MaxSAT instance by reusing the results of solving another similar MaxSAT instance.

imizes the number of unsatisfied clauses. In *partial MaxSAT*, the CNF formula $\phi = \phi_S \cup \phi_H$ contains a set of *soft clauses* $\phi_S$ and a set of *hard clauses* $\phi_H$. The goal is to find an assignment such that all hard clauses are satisfied while minimizing the number of unsatisfied soft clauses. Finally, a *weighted clause* is a pair $(c, w)$ where $w \in \mathbb{N}$ is the cost of not satisfying the clause $c$. In *weighted partial MaxSAT*, the goal is to find a total assignment where all hard clauses are satisfied, while minimizing the sum of the weights of unsatisfied soft clauses. In the remainder of the paper, we use MaxSAT to refer to the more general problem of weighted partial MaxSAT.

Most of the state-of-the-art MaxSAT algorithms rely on successive calls to a SAT solver. In particular, *Core-Guided* MaxSAT algorithms have been shown to be very effective in solving instances that arise from real-world applications [23]. These algorithms take advantage of the ability of SAT solvers to identify unsatisfiable subformulas (also known as *unsatisfiable cores*).

A SAT solver call $\texttt{SAT}(\phi, \mathcal{A})$ receives a CNF formula $\phi$ and a set of assumptions $\mathcal{A}$. The set $\mathcal{A}$ defines a set of literals that must be true in the model of $\phi$ returned by the SAT call. A SAT call returns a triple $(\mathsf{st}, \nu, \phi_C)$ where $\mathsf{st}$ denotes the solver status (SAT or UNSAT). If the call is satisfiable, then $\nu$ contains a model of $\phi$. Otherwise, $\phi_C \subseteq \phi$ contains a *core*: an unsatisfiable subformula of $\phi$. Note that a SAT call can return UNSAT, even when $\phi$ is satisfiable. This occurs when there is no model of $\phi$ such that all assumption literals in $\mathcal{A}$ can be set to true. In this case, $\phi_C$ contains clauses from $\phi$ as well as literals from $\mathcal{A}$.

## 3 Sequential Maximum Satisfiability

We define the *sequential MaxSAT problem* as the problem of solving a sequence of $n$ MaxSAT formulas $\phi^1, \phi^2, \ldots, \phi^n$, with $\phi^k \subseteq \phi^{k+1}$. This problem arises in many applications [7,16,18,28], where a sequence of MaxSAT instances are to be solved. In most cases, the $k$-th MaxSAT instance $\phi^k$ is generated by incrementally modifying the previous instance $\phi^{k-1}$, based on the solution of $\phi^{k-1}$.

A straightforward solution to the sequential MaxSAT problem is to use any off-the-shelf MaxSAT solver to *independently* solve each MaxSAT instance $\phi^k$ $(1 \leq k \leq n)$. This is the approach currently used in most applications [7,16,18,28]. However, this does not enable reusing information obtained from solving a given formula in solving the subsequent formulas.

This section is organized as follows. Section 3.1 reviews the *Fu & Malik* algorithm for MaxSAT with incremental SAT—previously published in [20]. Sections 3.2, 3.3 form the core contribution of the paper: Section 3.2 shows how to generalize *Fu & Malik* to solve sequential MaxSAT and Section 3.3 introduces restarts to cope with performance issues in the introduced algorithm.

### 3.1 Background: *Fu & Malik* MaxSAT Algorithm

The *Fu & Malik* algorithm [12] was initially proposed in 2006 and later extended to weighted MaxSAT [3,19]. More recently, a new version was proposed where the

**Algorithm 1:** Fu-Malik Algorithm with Incremental SAT [20]

```
   Input: φ = φ_H ∪ φ_S
   Output: optimal solution to φ
1  φ_W ← φ_H ∪ {c ∪ {blockingVar(c)} | c ∈ φ_S}      // fresh blocking variables
2  A ← {¬blockingVar(c) | c ∈ φ_S}                    // enable all soft clauses
3  while true do
4      (st, ν, φ_C) ← SAT(φ_W, A)
5      if st = SAT then return ν                      // optimal solution to φ
6      V_R ← ∅
7      m_C = min{weight(c) | c ∈ φ_C ∧ soft(c)}
8      foreach c ∈ φ_C ∧ soft(c) do
9          V_R ← V_R ∪ {r}                            // r is a fresh relaxation variable
10         c_r ← (c \ {blockingVar(c)}) ∪ {r} ∪ {b_r}  // b_r is a fresh variable
11         A ← A ∪ {¬b_r}                             // enable c_r
12         φ_W ← φ_W ∪ {c_r}
13         weight(c_r) ← m_C
14         if weight(c) > m_C then weight(c) ← weight(c) − m_C
15         else A ← (A \ {¬blockingVar(c)}) ∪ {blockingVar(c)}  // disable c
16     φ_W ← φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ 1)}
```

SAT solver is not rebuilt in each iteration, thus allowing the reuse of knowledge learnt by the SAT solver in previous iterations. Hence, the SAT solver is used incrementally for a *single* MaxSAT instance. Later we will extend this to use the whole MaxSAT solver incrementally, i.e. for multiple MaxSAT instances.

Algorithm 1 reviews the pseudo-code of *Fu & Malik* for solving weighted partial MaxSAT using SAT incrementally [20]. The working formula $\phi_W$ is initialized to all hard clauses with all soft clauses extended with a fresh *blocking variable*. Negations of the blocking variables are added to the assumptions $\mathcal{A}$, thus *enabling* the original soft clauses (lines 1-2). When a soft clause $c$ is extended with a blocking variable $b$ to form $(c \vee b)$, then adding $\neg b$ to the assumptions effectively enables $c$ since the SAT solver must necessarily satisfy $c$. Conversely, adding $b$ to the assumptions *disables* $c$ since $(c \vee b)$ is trivially satisfied.

Each iteration issues a SAT call on line 4. If the working formula is satisfiable, the optimal solution was found. Otherwise, $\phi_C$ is an unsatisfiable subformula (core). In this case, for each soft clause $c$ in $\phi_C$, a new relaxed clause $c_r$ is created from $c$ with two additional variables (a relaxation and a blocking variable). If the clause is enabled through the blocking variable, then the relaxation variable represents if the original clause is satisfied (or not) in the MaxSAT solution.

On line 7, the *weight of the core* $m_C$ is the minimum weight of all soft clauses in $\phi_C$. Soft clauses $c \in \phi_C$ with weight equal to $m_C$ are disabled (line 15) and replaced with their relaxation $c_r$. Soft clauses $c \in \phi_C$ with weight larger than $m_C$ are not removed. Their weight is decreased by $m_C$, thus resulting in a *clause split*, since the original weight is divided between $c$ and its relaxation $c_r$.

Finally, note that since the working formula is always expanded, the SAT solver is never rebuilt and its internal state is kept (including the learnt clauses).

### 3.2 Our Approach: Solving Sequential MaxSAT Incrementally

In this section we propose how to solve a sequential MaxSAT problem incrementally. Consider a sequence of MaxSAT formulas $\phi^1, \phi^2, \ldots, \phi^n$, with $\phi^k \subseteq \phi^{k+1}$.

We apply Algorithm 1 to $\phi^1$ and then extend the resulting working formula $\phi_W$ with hard clauses from $\phi_H^2 \setminus \phi_H^1$, and, soft clauses from $\phi_S^2 \setminus \phi_S^1$ each extended with a fresh blocking variable. Then resume the main loop of Algorithm 1 (from line 3). This process is analogously repeated for the upcoming formulas in the sequence. More precisely, each time $\phi_W$ becomes satisfiable, the clauses $\phi_H^{k+1} \setminus \phi_H^k$, and $\phi_S^{k+1} \setminus \phi_S^k$ are added to $\phi_W$, where the soft clauses are adorned with a fresh blocking variable, which is in turn reflected in the assumptions. Then go to line 3.

As such, it is not necessary to restart the search from scratch for each formula in the sequence. This approach is correct because the addition of new soft or hard clauses does not invalidate any of the previously found cores. Note that the approach is incremental at two levels: it uses the SAT solver incrementally for each instance but also is incremental across the sequence of instances.

### 3.3 Extending Sequential MaxSAT Solving with Restarts

Consider a sequence of MaxSAT instances $\phi^1, \ldots, \phi^n$ where $\phi^i \subseteq \phi^j$ for $1 \leq i < j \leq n$. When solving $\phi^i$ first, the incremental *Fu & Malik* has a "narrower perspective" than the non-incremental *Fu & Malik* applied directly on $\phi^j$. More specifically, the set of possible cores in $\phi^i$ is always a subset of the possible cores in $\phi^j$. Consequently, the incremental version may end up finding a core of *poorer quality* than the non-incremental version. Finding a core of poor quality is often detrimental to the rest of the computation. This is especially true for the weighted *Fu & Malik*, which splits clauses based on the minimum weight of the found core. This is illustrated by Example 1.

*Example 1.* Consider an $n \in \mathbb{N}$, weights $w_1 < w_2 \in \mathbb{N}$, and the core $\mathcal{C}[w_1, w_2] = \{(w_2, \neg a_i) \mid i \in 1..n\} \cup \{(w_1, b \vee \bigvee_{i \in 1..n} a_i), (w_2, \neg b)\}$. Once found, this core is conceptually split into the sets of clauses $\mathcal{C}[w_1, w_1]$ and $\mathcal{C}[0, w_2 - w_1]$, where the first set is relaxed. This creates $n + 1$ new clauses and relaxation variables, incurring thus cost on further computation. If the next iteration adds the hard clause $(b)$, then the MaxSAT solver can use the simpler core $\{(b), (w_2, \neg b)\}$ without encountering the large core above.

Here we propose a solution to the above-outlined issue, which is to *restart* the whole computation once we suspect that the incremental version is finding cores of poor quality. We say that a given soft clause $c \in \phi_C$ is *split* if its weight is larger than the weight of the unsatisfiable core $(m_C)$. If a clause $c$ is split, it means that an unsatisfiable core with other soft clauses with smaller weights was found. In our solver, we maintain a *split counter* for every soft clause in the formula and define a split limit. When the split limit is reached for some soft clause, the solver is rebuilt and Algorithm 1 is restarted. In order to maintain completeness, the solver restarts at most once for each MaxSAT formula $\phi^i$ in the sequential MaxSAT instance.

## 4 Empirical Evaluation

We evaluate our technique on sequential MaxSAT problems generated from three applications: abstraction refinement, user-guided analysis, and statistical relational inference. Abstraction refinement [28] tackles a central problem in software verification: finding a program abstraction that only tracks information relevant to proving assertions of interest. It solves a sequence of MaxSAT instances to construct such an abstraction. User-guided analysis [18] iteratively incorporates user feedback in software analysis tools to eliminate false alarms. In each iteration, it solves a MaxSAT instance to infer the most likely set of true alarms based on the current feedback. Statistical relational inference [16] enables a wide range of information retrieval tasks by solving a system of weighted first-order constraints over a relational database. It scales to large database instances by lazily solving a sequence of progressively growing MaxSAT instances.

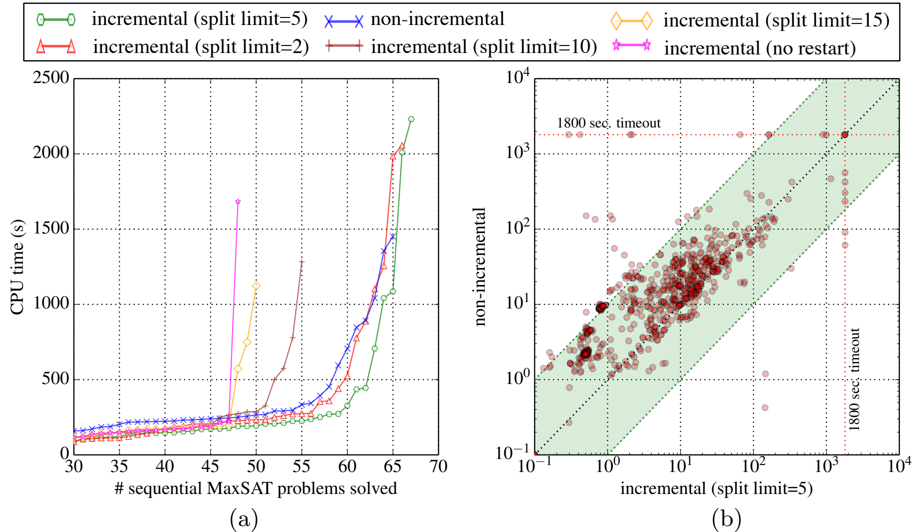
We implemented our technique in the Open-WBO [22] MaxSAT solver. All experiments were done on a Linux machine with a 3.0 GHz processor. We limited each MaxSAT solver invocation to 32 GB RAM and 30 minutes of CPU time.

We compare our incremental algorithm with restarts to two baselines: the non-incremental and the incremental-without-restarts algorithms. The former is the original Open-WBO solver while the latter is obtained by disabling restarts in our solver. To evaluate the effect of different split limits, we use the split limits 2, 5, 10, and 15. We generated the sequential MaxSAT instances by running the applications with both our solver (using split limit of 5) and the non-incremental solver until the application terminates or any MaxSAT invocation exceeds one hour. Since the solutions returned by the MaxSAT solver may affect the sequence of MaxSAT instances generated by the applications, we used both the solvers to reduce the bias introduced by a particular solver in the instance generation. Following this recipe, we obtained 74 sequential MaxSAT problems comprising 669 MaxSAT instances. The number of clauses in each MaxSAT instance ranges from two thousand to 150 million, with 10 million being the average.

The cactus plot in Figure 1(a) shows the number of sequential MaxSAT instances solved by our approaches and the baseline approaches within given CPU times. As the plot shows, our incremental algorithm with 5 as the split limit solves the most instances.

Moreover, on the instances that can be solved by both approaches, our approach with 5 as the split limit yields an average speedup of 1.8× over the non-incremental approach. On certain instances, the benefit is as high as 4.7×. The scatter plot in Figure 1(b) further compares the time consumed by both approaches on each individual MaxSAT instance. As the plot shows, the speedup can be as high as 296× on certain instances. This shows that our approach effectively improves the overall performance by reusing computation across similar MaxSAT instances in the same sequence.

We also observe that the incremental algorithm without restarts performs significantly worse compared to other approaches. This justifies the need for restarts in incremental MaxSAT solving: naively reusing cores computed from previous smaller instances can severely impede the solver's performance on the




(a) (b)

**Fig. 1.** Performance of our approaches and baseline approaches on (a) sequential MaxSAT problems and (b) each individual MaxSAT instance in the sequences.

current instance. On the other hand, our approach effectively avoids this problem by restarting the solving process when it observes any clause being split too often.

We further observe that using a split limit that is too high (e.g., 10 or 15) or too low (e.g., 2) adversely affects the performance of the incremental algorithm. When the cores learnt from previous smaller MaxSAT instances are unsuitable for the current MaxSAT instance, a too high split limit can either fail to trigger the restart or only triggers the restart after the algorithm has spent significant time running with these cores. On the other hand, using a too low limit can trigger the restart too often, making the algorithm fall back to its non-incremental version. While finding an adequate restart condition is an interesting research direction, using 5 as the split limit yields the best overall performance on the evaluated instances.

## 5 Discussion and Future Work

Incrementality and restarts are well established in SAT solving [10, 11], so a natural question is why they do *not* directly translate to MaxSAT. Adding new clauses to a SAT solver does not invalidate existing learnt clauses just as new clauses do not invalidate existing cores in a MaxSAT solver. Yet, core reuse leads to a decline in performance in MaxSAT (see Sec. 4). This reveals the inherent issue of computing *cores of poor quality* when solving the smaller instance (see Example 1). In SAT, poor quality clauses from previous computations are eventually deleted. In MaxSAT, poor quality cores can be detrimental to the rest of the computation. This is especially true for the weighted *Fu & Malik* algorithm, which creates new clauses by splitting [19]. Bad quality cores are also known to arise in the standard formulation of weighted MaxSAT. There are approaches to resolve the issue, namely *stratification* [2] and *formula partitioning* [21], which

iteratively consider subformulas of the original formula. Note that the same ideas cannot be easily adapted to our setting since the complete MaxSAT formula in our case is not available to the algorithm in advance. Also note that although stratification can be applied to separate MaxSAT instances in the sequence, many of them are satisfiable, which results in stratification being inefficient in practice, as also confirmed by our experience.

The proposed approach uses two levels of incrementality at the same time: (1) it uses incremental SAT calls inside a MaxSAT solver and (2) it makes MaxSAT calls also incremental. This means that all the information learnt during the sequential problem solving is kept until the problem is solved completely. Although the standard way to solve a sequence of MaxSAT instances is to restart a MaxSAT solver at each iteration while doing incremental SAT calls inside, alternatively one could consider using incrementality only for the MaxSAT calls instead. For this, one needs to keep all unsatisfiable cores computed at each preceding MaxSAT call, relax the corresponding clauses of the formula, and reconstruct the cardinality constraints.

Observe that the proposed ideas cannot be easily applied to algorithms that are *not* core-guided. In the classical SAT-UNSAT, UNSAT-SAT linear and binary search MaxSAT algorithms, the SAT solver might learn constraints that are invalid for solving the next MaxSAT formula, so it would have to be restarted for each MaxSAT formula in the sequence. Also note that the *Fu & Malik* algorithm has the drawback of relaxing clauses more than once and thus introducing many auxiliary variables. Therefore, it is of great interest to adapt the proposed ideas to more recent MaxSAT algorithms that resolve this issue (e.g. [1,4–6,8,9,14,15,24–26]). An immediate improvement of the proposed approach would be to devise more fine-grained restart strategies, that is, selectively keeping certain good cores, instead of completely restarting from scratch. Finally, it is also interesting to explore incrementality when clauses are not only added but also deleted.

## 6 Conclusion

This paper explores an incremental approach to core-guided MaxSAT solving. We begin by extending a core-guided MaxSAT algorithm for sequences of instances where clauses are gradually added. Experimental evaluation shows that this approach in fact yields *worse* performance than applying the MaxSAT solver on each instance from scratch. This is due to the inherent problem of learning "bad" information from instances earlier in the sequence. We propose *restarts* which enable discarding learnt information if deemed unuseful. Our restart strategy significantly outperforms the non-incremental version.

**Acknowledgments.** *This work was supported by the national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, DARPA under agreement #FA8750-15-2-0009, NSF awards #1253867 and #1526270, and a Facebook Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.*

## References


1. Alviano, M., Dodaro, C., Ricca, F.: A MaxSAT algorithm using cardinality constraints of bounded size. In: IJCAI (2015)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: CP 2012 (2012)
3. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial MaxSAT through satisfiability testing. In: SAT (2009)
4. Ansótegui, C., Bonet, M.L., Levy, J.: SAT-based MaxSAT algorithms. Artif. Intell. 196 (2013)
5. Ansótegui, C., Didier, F., Gabàs, J.: Exploiting the structure of unsatisfiable cores in MaxSAT. In: IJCAI (2015)
6. Bjorner, N., Narodytska, N.: Maximum satisfiability using cores and correction sets. In: IJCAI (2015)
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: A modular approach to MaxSAT modulo theories. In: SAT (2013)
8. Davies, J., Bacchus, F.: Solving MAXSAT by solving a sequence of simpler SAT instances. In: CP (2011)
9. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: SAT (2013)
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT (2003)
11. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
12. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: SAT (2006)
13. Grigore, R., Yang, H.: Abstraction refinement guided by a learnt probabilistic model. In: POPL (2016)
14. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: AAAI (2011)
15. Ignatiev, A., Morgado, A., Manquinho, V.M., Lynce, I., Marques-Silva, J.: Progression in maximum satisfiability. In: ECAI (2014)
16. Mangal, R., Zhang, X., Kamath, A., Nori, A.V., Naik, M.: Scaling relational inference using proofs and refutations. In: AAAI (2016)
17. Mangal, R., Zhang, X., Nori, A., Naik, M.: Volt: A lazy grounding framework for solving very large MaxSAT instances. In: SAT (2015)
18. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: FSE (2015)
19. Manquinho, V.M., Marques-Silva, J.P., Planes, J.: Algorithms for weighted boolean optimization. In: SAT (2009)
20. Martins, R., Joshi, S., Manquinho, V.M., Lynce, I.: Incremental cardinality constraints for MaxSAT. In: CP (2014)
21. Martins, R., Manquinho, V.M., Lynce, I.: On partitioning for maximum satisfiability. In: ECAI 2012 (2012)
22. Martins, R., Manquinho, V.M., Lynce, I.: Open-WBO: A modular MaxSAT solver. In: SAT (2014)
23. MaxSAT evaluations, http://www.maxsat.udl.cat/
24. Morgado, A., Dodaro, C., Marques-Silva, J.: Core-guided MaxSAT with soft cardinality constraints. In: CP (2014)
25. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: A survey and assessment. Constraints 18(4) (2013)

26. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided MaxSAT resolution. In: AAAI (2014)
27. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62(1-2) (2006)
28. Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H.: On abstraction refinement for program analyses in Datalog. In: PLDI (2014)
29. Zhang, X., Mangal, R., Nori, A.V., Naik, M.: Query-guided maximum satisfiability. In: POPL (2016)