Incremental Maximum Satisfiability

Andreas Niskanen
HIIT, Department of Computer Science, University of Helsinki, Finland

Jeremias Berg
HIIT, Department of Computer Science, University of Helsinki, Finland

Matti Järvisalo
HIIT, Department of Computer Science, University of Helsinki, Finland

Abstract

Boolean satisfiability (SAT) solvers allow for incremental computations, which is key to efficient employment of SAT solvers iteratively for developing complex decision and optimization procedures, including maximum satisfiability (MaxSAT) solvers. However, enabling incremental computations on the level of constraint optimization remains a noticeable challenge. While incremental computations have been identified to have great potential in speeding up MaxSAT-based approaches for solving various real-world optimization problems, enabling incremental computations in MaxSAT remains to most extent unexplored. In this work, we contribute towards making incremental MaxSAT solving a reality. Firstly, building on the IPASIR interface for incremental SAT solving, we propose the IPAMIR interface for implementing incremental MaxSAT solvers and for developing applications making use of incremental MaxSAT. Secondly, we expand our recent adaptation of the implicit hitting set based MaxHS MaxSAT solver to a fully-fledged incremental MaxSAT solver in terms of implementing the IPAMIR specification in full, and detail in particular how, in addition to weight changes, assumptions are enabled without losing incrementality. Thirdly, we provide further empirical evidence on the benefits of incremental MaxSAT solving under assumptions.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Constraint and logic programming

Keywords and phrases maximum satisfiability, MaxSAT, incremental optimization, API, implicit hitting set approach

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.14

Supplementary Material Source code and experiment data available at `https://bitbucket.org/coreo-group/incremental-maxhs/`.

Funding Work financially supported by Academy of Finland under grants 322869 and 342145.

Acknowledgements The authors wish to thank Fahiem Bacchus, Alexey Ignatiev, Ruben Martins, and Peter Stuckey for valuable discussions on IPAMIR, and the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Beyond one-shot search for satisfiability, incremental use of of Boolean satisfiability (SAT) solvers [12, 22] is a key contributing factor in the successful employment of SAT solvers as "practical NP-oracles" in a range of applications and as the basis for developing complex decision, search and optimization procedures [4, 17, 13, 19, 11]. Incrementality allows for maintaining solver state, including learned clauses and heuristic scores, between consecutive solver calls on related SAT instances, and altering the instance at hand by adding clauses and forcing a partial assignment for the next solver call through so-called assumptions. The IPASIR application programming interface (API) has been developed as a standard for building incremental applications of SAT [5].

Maximum satisfiability (MaxSAT) [4] is the natural optimization extension of SAT. The constraints underlying an optimization problem are encoded in MaxSAT as "hard" propositional clauses, which all solutions must satisfy. The objective function at hand is encoded using weighted "soft" clauses, with the weights standing for the objective function coefficients, the aim being to find an optimal solution, i.e., an assignment that minimizes the sum of the weights of the soft clauses it falsifies. MaxSAT is one of the successes of incremental SAT solving: most if not all modern MaxSAT solvers make heavy use of incremental SAT, typically using assumptions for iteratively extracting unsatisfiable cores, i.e., subsets of the set of soft clauses that together with the hard clauses are unsatisfiable [29, 16, 9, 26, 1, 2]. However, in various real-world settings *incremental MaxSAT solving* would be beneficial for more efficiently finding optimal solutions to a sequence of related MaxSAT instances [20, 7, 33, 30]. For example, in different types of timetabling problems, changes to the available resources (such as e.g. availability of classrooms, changes to the curriculum and the like in university timetabling) may render a previously obtained optimal timetabling invalid, and it would be beneficial to be able to incrementally compute a new optimal solution [18, 10]. The availability of resources and other changes in constraints would require the ability to add or remove previous hard or soft clauses, or the ability to call a MaxSAT solver with different assumptions. As another example, changes to the weights of soft clauses would be beneficial to handle incrementally; one application scenario for this is in the context of learning classifiers where MaxSAT has been employed for implementing AdaBoost [15].

Although achieving truly incremental computations in declarative optimization remains a challenge, MaxSAT is a promising paradigm when it comes to achieving high levels of incrementality, in particular due to many state-of-the-art solvers performing unsatisfiability-based search (via iterative unsatisfiable core extraction) instead of solution-improving search towards finding better and better solutions. However, despite the potential of incremental MaxSAT solving, so far only partial solutions in terms to "true" incrementality have been proposed [32, 27].

In this work, we make several contributions towards making incremental MaxSAT solving a reality. Firstly, we outline the various forms of incrementality called for in MaxSAT solving, and—extending on IPASIR for incremental SAT—propose an API for incremental MaxSAT. The API, named IPAMIR, provides a common interface for implementing support for incremental computations in MaxSAT solvers as well as for application developers making use of future incremental MaxSAT solvers. The interface is also at center stage in MaxSAT Evaluation 2022 (`https://maxsat-evaluations.github.io/2022/`), with the introduction of the incremental track. Secondly, we develop what we believe to be a first openly-available incremental MaxSAT solver in its generality. The solver supports all functionality specified in IPAMIR. Extending further a recent adaptation of the successful MaxHS implicit hitting set (IHS) style MaxSAT solver [8, 6] that so-far supports changing weights of soft clauses incrementally [27], we integrate further functionalities to the solver, including support for incrementally handling MaxSAT solver calls under assumptions. The choice of extending MaxHS in particular is motivated by the fact that the IHS approach does not alter the input formula during computations like the so-called core-guided approaches do, which makes it a prime candidate for developing support for incrementality. Thirdly, complementing previous evidence on the benefits of supporting incrementality under changes to soft clause weights [27], we provide empirical evidence that support for incremental computations under different sets of assumptions is similarly promising in terms of speeding up solving of sequences of interrelated MaxSAT instances.

2 Maximum Satisfiability

For a Boolean variable x, there are two literals x and $\neg x$. For a set L of literals, the set $\neg L$ consists of their negations. A clause C is a disjunction of literals and a CNF formula F a conjunction of clauses. We will mostly view a clause as a set of literals and a formula as a set of clauses. The set of variables of a clause C and a CNF formula F are $\mathtt{var}(C) = \{x \mid x \in C \text{ or } \neg x \in C\}$ and $\mathtt{var}(F) = \bigcup_{C \in F} \mathtt{var}(C)$, respectively. The set $\mathtt{lit}(F)$ of literals of a CNF formula F is $\mathtt{lit}(F) = \bigcup_{C \in F} C$. A (truth) assignment τ maps Boolean variables to 1 (true) or 0 (false). Truth assignments extend to literals l, clauses C and formulas F in the standard way: $\tau(\neg l) = 1 - \tau(l)$, $\tau(C) = \max\{\tau(l) \mid l \in C\}$ and $\tau(F) = \min\{\tau(C) \mid C \in F\}$.

An instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_S, w)$ of (weighted partial) MaxSAT consists of two CNF formulas, the hard clauses $\mathcal{F}_H$ and the soft clauses $\mathcal{F}_S$, and a weight function $w \colon \mathcal{F}_S \to \mathbb{N}$ that assigns a positive weight to each soft clause. An assignment τ that satisfies $\mathcal{F}_H$ (i.e., for which $\tau(\mathcal{F}_H) = 1$ is a solution of $\mathcal{F}$, its cost $\textsc{cost}(\mathcal{F}, \tau) = \sum_{C \in \mathcal{F}_S} w(C)(1 - \tau(C))$ is the sum of weights of the soft clauses it falsifies. A solution τ is optimal if $\textsc{cost}(\mathcal{F}, \tau) \leq \textsc{cost}(\mathcal{F}, \tau')$ holds for any solution τ' of $\mathcal{F}$. The cost of optimal solutions of $\mathcal{F}$ is denoted by $\textsc{cost}(\mathcal{F})$. When convenient, we treat a solution τ as the set of literals the assignment satisfies, i.e, as $\tau = \{l \mid \tau(l) = 1\}$.

Normalized Form

From now on, we will assume that all MaxSAT instances are in what we call normalized form. Specifically, for a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_S, w)$, we assume each soft clause $C \in \mathcal{F}_S$ is a unit clause of form $C = (\neg b)$ containing the negation of a literal b. The assumption can be made w.l.o.g. since any soft clause C can be extended with a fresh variable $b \notin \mathtt{var}(\mathcal{F}_H \wedge \mathcal{F}_S)$ (depending on the context, sometimes called a relaxation variable, reification variable, blocking variable, or assumption variable in the literature) to form the hard clause $C \vee b$ and the soft clause $(\neg b)$. The set $\mathcal{F}_L$ of soft literals contains all literals b for which $(\neg b) \in \mathcal{F}_S$. By extending the weight function w to soft literals by $w(b) = w((\neg b))$ the cost of a solution τ can equivalently be expressed as $\textsc{cost}(\mathcal{F}, \tau) = \sum_{b \in \mathcal{F}_L} \tau(b) w(b)$. Instead of the classical way of viewing MaxSAT in terms of hard and soft clauses, we will from now on adopt the equivalent view of treating a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$ as a set $\mathcal{F}_H$ of hard clauses, a set $\mathcal{F}_L$ of soft literals and a weight function $w \colon \mathcal{F}_L \to \mathbb{N}$.

MaxSAT under Assumptions

A set A of assumptions is a set of literals such that $x \notin A$ or $\neg x \notin A$ holds for any variable x. Solving a MaxSAT instance $\mathcal{F}$ under a set of assumptions refers to computing an optimal solution to the MaxSAT instance $\mathcal{F} \wedge A = (\mathcal{F}_H \wedge \bigwedge_{l \in A}(l), \mathcal{F}_L, w)$.

▶ **Example 1.** Consider the MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$ with $\mathcal{F}_H = \{(b_1 \vee x), (\neg x \vee b_2), (\neg z), (z \vee y \vee b_3 \vee b_4), (\neg y \vee b_3 \vee b_4)\}$, $\mathcal{F}_L = \{b_1, b_2, b_3, b_4\}$ and $w(b_1) = w(b_3) = w(b_4) = 1$ and $w(b_2) = 2$. The solution $\tau = \{b_1, \neg b_2, b_3, \neg b_4, \neg x, y, \neg z\}$ is an optimal solution of $\mathcal{F}$, with $\textsc{cost}(\mathcal{F}, \tau) = \textsc{cost}(\mathcal{F}) = 2$.

3 Incremental MaxSAT

The goal of incremental MaxSAT solving is to solve a sequence of related MaxSAT instances. The input to an incremental MaxSAT solver is a sequence of MaxSAT instances $(\mathcal{F}^1, \ldots, \mathcal{F}^k)$.

However, this sequence need not be predefined, and instead, a next instance may be adaptively formed: for all $i = 1, \ldots, k-1$ the instance $\mathcal{F}^{i+1}$ may depend on the optimal solutions $\tau_1, \ldots, \tau_i$ of the previous instances in the sequence.

A next instance $\mathcal{F}^{i+1}$ is obtained from $\mathcal{F}^i$ by applying a set of changes to $\mathcal{F}^i$. We consider the following types changes to a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$.

- *Adding hard clauses.* Given a clause C, add it to the hard clauses of $\mathcal{F}$: $\textsc{addHard}(\mathcal{F}, C) = (\mathcal{F}_H \cup \{C\}, \mathcal{F}_L, w)$.
- *Adding soft literals.* Given a variable $b \notin \mathcal{F}_L$ with weight w_b, add it to the set of soft literals of $\mathcal{F}$: $\textsc{addSoft}(\mathcal{F}, b, w_b) = (\mathcal{F}_H, \mathcal{F}_L \cup \{b\}, w \cup \{b \mapsto w_b\})$.
- *Changing the weight of a soft literal.* Given a literal $b \in \mathcal{F}_L$ with weight w_b, change its weight in $\mathcal{F}$ to w_b: $\textsc{changeWeight}(\mathcal{F}, b, w_b) = (\mathcal{F}_H, \mathcal{F}_L, (w \setminus \{b \mapsto w(b)\}) \cup \{b \mapsto w_b\})$.

Let us shortly describe a few use cases for these types of changes. MaxSAT-based CEGAR algorithms (e.g. [21, 28]) iteratively add hard or soft clauses in order to refine the search space. Recalling that a soft clause C may be added via the hard clause $C \vee b$ with b as a soft literal, addHard and addSoft cover these types of applications. In turn, changeWeight has proven useful e.g. in implementing MaxSAT-based AdaBoost [15, 27].

Any number of these changes may be performed to transform $\mathcal{F}^i$ to $\mathcal{F}^{i+1}$. In addition, we allow for enforcing *assumptions* on the variables of an instance. This means that, in addition to $\mathcal{F}^i$, at each iteration the input instance contains given a set of assumption literals A^i. The goal at iteration i to solve the instance $\mathcal{F}^i \wedge A^i$. Note that assumptions allow for simulating the removal of hard clauses and hardening soft clauses. Removing a soft literal can in turn be simulated by setting its weight to zero.

▶ **Example 2.** Consider the MaxSAT instance $\mathcal{F}$ from Example 1. Suppose we solve it under the assumptions $A = \{x\}$, that is, enforcing that $\tau(x) = 1$ must hold for any solution of $\mathcal{F}$. Now $\tau = \{\neg b_1, b_2, b_3, \neg b_4, x, y, \neg z\}$ is an optimal solution of $\mathcal{F}$ under the assumptions A, with $\textsc{cost}(\mathcal{F} \wedge A, \tau) = 3$. Note that if we view the hard clause $b_1 \vee x$ as a normalized soft clause x with weight $w(b_1)$, by assuming $\{\neg b_1\}$ we effectively *harden* the soft clause x, which in this case achieves the same result as assuming $\{x\}$. Now suppose we instead consider the instance $\mathcal{F}' = \textsc{changeWeight}(\mathcal{F}, b_1, 0)$ which sets the weight of the first soft literal to zero. An optimal solution of $\mathcal{F}'$ is $\tau' = \{b_1, \neg b_2, b_3, \neg b_4, \neg x, y, \neg z\}$ with cost $\textsc{cost}(\mathcal{F}', \tau') = 1$. Note how assigning its weight to 0 effectively removes b_1 as a soft literal.

4 IPAMIR: Interface for Incremental MaxSAT Solving

We continue by describing IPAMIR (Re-entrant Incremental MaxSAT solver API), as our proposal for a generic interface for incremental MaxSAT solving. We build heavily on IPASIR [5], the standard interface for incremental SAT solving. The IPAMIR interface is available in open source at `https://bitbucket.org/coreo-group/ipamir/`. Examples of its usage are also provided in the repository in the `app` directory.

The functions specified by IPAMIR are listed in Figure 1. The functions `ipamir_init`, `ipamir_release`, `ipamir_assume`, `ipamir_solve`, and `ipamir_set_terminate` are identical to their IPASIR counterparts [5]. Generalizing on IPASIR and its `ipasir_add` function, we distinguish between adding hard clauses via `ipamir_add_hard` and soft literals via `ipamir_add_soft_lit`. Note that as in IPASIR, `ipamir_add_hard` adds literals into a clause one at a time, and the current clause is finalized with zero (as in the DIMACS formats). The `ipamir_add_soft_lit` function declares the given literal as a soft literal with the given weight, represented as a 64-bit unsigned integer, implementing addSoft. If the

```
// Construct a MaxSAT solver and return a pointer to it.
void * ipamir_init ();
// Deallocate all resources of the MaxSAT solver.
void ipamir_release (void * solver);
// Add a literal to a hard clause or finalize the clause with zero.
void ipamir_add_hard (void * solver, int32_t lit_or_zero);
// Add a weighted soft literal.
void ipamir_add_soft_lit (void * solver, int32_t lit, uint64_t weight);
// Assume a literal for the next solver call.
void ipamir_assume (void * solver, int32_t lit);
// Solve the MaxSAT instance under the current assumptions.
int ipamir_solve (void * solver);
// Compute the cost of the solution.
uint64_t ipamir_val_obj (void * solver);
// Extract the truth value of a literal in the solution.
int32_t ipamir_val_lit (void * solver, int32_t lit);
// Set a callback function for terminating the solving procedure.
void ipamir_set_terminate (void * solver, void * state,
                           int (*terminate)(void * state));
```

Figure 1 Functions declared in the IPAMIR header.

argument literal in question has already been declared soft, `ipamir_add_soft_lit` changes its weight to the given weight, thereby also implementing CHANGEWEIGHT. We declare the function `ipamir_val_obj` to extract the cost, i.e., the value of the objective function, of the current solution, and `ipamir_val_lit` to extract the value of a literal, corresponding to `ipamir_val` in IPASIR.

The function `ipamir_solve` invokes a MaxSAT solver on a MaxSAT instance defined by calls to `ipamir_add_hard` and `ipamir_add_soft_lit` under the current assumptions defined by calls to `ipamir_assume`. Upon termination it returns a code corresponding to the state of the solver. If the search is interrupted before determining whether a solution exists, the state of the solver is changed to INPUT and `ipamir_solve` returns 0. If no solution exists, the state of the solver is changed to UNSAT and `ipamir_solve` returns 10. If the search is interrupted via `ipamir_set_terminate` but a solution (not necessarily optimal) has been found, the state of the solver is changed to SAT and `ipamir_solve` returns 20. If an optimal solution is found, the state of the solver is changed to OPTIMAL and `ipamir_solve` returns 30. Finally, if the solver is in the state ERROR, `ipamir_solve` returns 40. The solver enters state ERROR if a sequence of IPAMIR function calls have been made which the solver does not support (e.g., using assumptions, changing weights of soft literals). The state transitions are depicted in Figure 2. Note that in contrast to IPASIR [5], we distinguish between OPTIMAL and SAT. Furthermore, as a current design choice, the dedicated ERROR state is included in order to accommodate solvers which do not implement IPAMIR in full.

5 Incremental Implicit Hitting Set based MaxSAT Solving

We describe an extension of the implicit hitting set based MaxSAT algorithm that implements the IPAMIR interface in full in an incremental way. In particular, we build on the recent extension of the state-of-the-art IHS solver MaxHS [8, 6] to support incremental computation under changes to weights [27], to allow incremental computations w.r.t. all functionality of IPAMIR. A major aspect of this extension is enabling incremental MaxSAT solving under assumptions. We start with an overview of the IHS approach to MaxSAT solving to the extent necessary, and then detail how we adapt it to incremental computations under

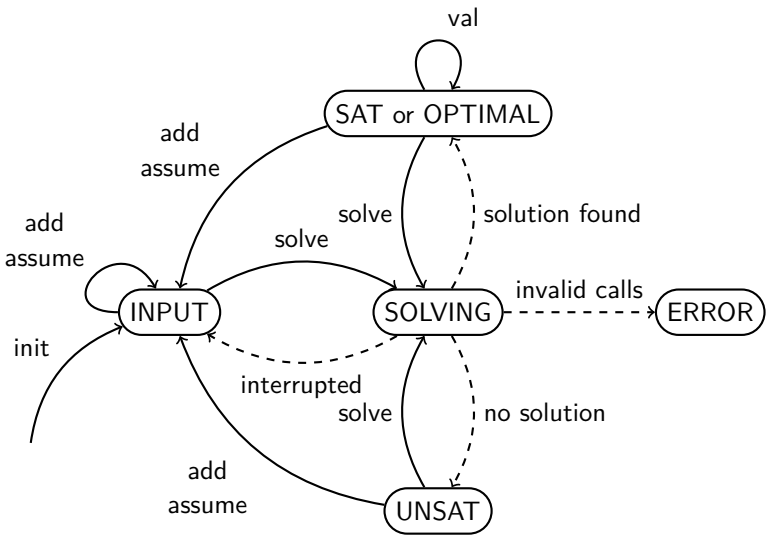

Figure 2 IPAMIR solver states and state transitions.

assumptions and what implementation-level changes to MaxHS are necessary for achieving full incrementality.

5.1 The Implicit Hitting Set Approach to MaxSAT

The IHS algorithm for MaxSAT makes use of unsatisfiable cores and hitting sets. For a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$, a (an unsatisfiable) core is a clause C that is entailed by $\mathcal{F}_H$ and for which $\texttt{lit}(C) \subset \mathcal{F}_L$. In other words, C is a core if it is satisfied by all solutions of $\mathcal{F}$ and only contains soft literals of $\mathcal{F}$. Note that this definition is equivalent to the more standard one of a core being a set of soft clauses which is unsatisfiable together with the hard clauses, since $\mathcal{F}_H \wedge \bigwedge_{(\neg b)\in\kappa}(\neg b)$ is unsatisfiable for a subset $\kappa \subset \mathcal{F}_S$ if and only if $\mathcal{F}_H$ entails the clause $\bigvee_{(\neg b)\in\kappa} b$. (Our definition actually better captures how cores are expressed in MaxSAT solvers making use of cores.)

For a set $\mathcal{C}$ of cores, a hitting set $hs \subset \mathcal{F}_L$ is a set of soft literals that has non-empty intersection with each $\kappa \in \mathcal{C}$ (each κ viewed as a set of literals). The cost of a hitting set hs is $\textsc{cost}(\mathcal{F}, hs) = \sum_{b\in hs} w(b)$. A hitting set hs^m is minimum-cost if $\textsc{cost}(\mathcal{F}, hs^m) \leq \textsc{cost}(\mathcal{F}, hs)$.

IHS MaxSAT solvers [8, 31, 6] rely on the well-known fact that hitting sets over sets of cores of a MaxSAT instance provide lower bounds on the optimal cost of the instance. More formally, let hs be a minimum-cost hitting set over a set $\mathcal{C}$ of cores of an instance $\mathcal{F}$. Then $\textsc{cost}(\mathcal{F}, hs) \leq \textsc{cost}(\mathcal{F})$.

Algorithm 1 details `IHS`, a basic implicit hitting set algorithm to computing an optimal solution to a given MaxSAT instance $\mathcal{F}$. The algorithm alternates between core extraction (the `Extract-Cores` subroutine) and hitting-set computation (the `Min-Hs` subroutine). The former extracts cores of $\mathcal{F}$ and accumulates them in the set $\mathcal{C}$. Upon termination, the `Extract-Cores` subroutine also returns a (not necessarily optimal) solution τ of $\mathcal{F}$ that is used to refine the upper bound ub on $\textsc{cost}(\mathcal{F})$. The latter subroutine computes a minimum-cost hitting set over $\mathcal{C}$. The cost $\textsc{cost}(\mathcal{F}, hs)$ of such a hitting set hs is a lower bound on $\textsc{cost}(\mathcal{F})$. The algorithm terminates once $lb = ub$ and returns τ_{best}, the found solution for which $\textsc{cost}(\mathcal{F}, \tau_{best}) = ub$ and which is guaranteed to be an optimal solution.

```
1  IHS(F)
     Input: An instance F = (F_H, F_L, w)
     Output: An optimal solution τ
2    lb ← 0; ub ← ∞;
3    τ_best ← ∅; C ← ∅;
4    while (TRUE) do
5      hs ← Min-Hs(F_L, C);
6      lb = COST(F, hs);
7      if (lb = ub) then break;
8      (K, τ) ← Extract-Cores(F_H, F_L, hs);
9      if (COST(F, τ) < ub) then
         τ_best ← τ; ub ← COST(F, τ);
10     if (lb = ub) then return τ_best;
11     C ← C ∪ K;
```

Algorithm 1 MaxSAT solving via implicit hitting sets

$$\textbf{minimize} \quad \sum_{b \in \mathcal{F}_L} w(b) \cdot b$$

$$\textbf{subject to}$$

$$\sum_{b \in \kappa} b \geq 1 \qquad \forall \kappa \in \mathcal{C}$$

$$b \in \{0, 1\} \qquad \forall b \in \mathcal{F}_L$$

Figure 3 An integer program for computing a hitting set over a set $\mathcal{C}$ of cores of an instance $\mathcal{F}$

In more detail, on input $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$ the upper and lower bounds ub and lb are initialized to ∞ and 0, respectively, on Line 2. Further, the best known model τ_{best} is initialized to $\varnothing$ and a set $\mathcal{C}$ of cores of $\mathcal{F}$ (represented as sets of soft literals) to $\emptyset$ on Line 3. The main search loop (Lines 4-11) iterates while $lb < ub$. During each iteration of the loop, a minimum-cost hitting set hs over $\mathcal{C}$ is computed on Line 5 by solving the integer program detailed in Figure 3 via the procedure `Min-Hs`$(\mathcal{F}_L, \mathcal{C})$, representing negative soft literals $\neg x \in \mathcal{F}_L$ in the standard way as the term $(1 - x)$. The cost $\textsc{cost}(\mathcal{F}, hs)$ is used for updating the lower bound lb on $\textsc{cost}(\mathcal{F})$ on Line 6. Since no cores are removed from $\mathcal{C}$ during the execution of `IHS`, $\textsc{cost}(\mathcal{F}, hs)$ is non-decreasing over the iterations.

After updating the lower bound, the termination criterion is checked on Line 7. If $ub = lb$, the algorithm terminates and returns the current best solution τ_{best} as optimal. Otherwise, the core extraction step `Extract-Cores` is invoked on Line 8. The procedure employs the assumption interface offered by most modern SAT solvers to extract previously unseen cores of $\mathcal{F}$ in the form of a disjoint set K of cores such that each $\kappa \in K$ is a subset of $\mathcal{F}_L \setminus hs$. When no more such cores can be found, the SAT solver also provides a solution τ of $\mathcal{F}$. The current upper bound ub is updated to the cost $\textsc{cost}(\mathcal{F}, \tau)$ if applicable on Line 9. If the updated bounds match, the algorithm terminates on Line 10. Otherwise, the new cores in K are added to $\mathcal{C}$ and the loop reiterated.

An important intuition for understanding the `IHS` algorithm is that all cores in K are disjoint from the hitting set hs and are thus not hit by hs. Adding the new cores to $\mathcal{C}$ results in hs not being a hitting set over $\mathcal{C}$ in subsequent iterations. The algorithm terminates in the worst-case after having accumulated all cores of the input instance.

▶ **Example 3.** Consider an invocation of `IHS` on the MaxSAT instance $\mathcal{F}$ from Example 1. Initially $\mathcal{C} = \emptyset$ so the first call to `Min-Hs` returns $hs = \emptyset$ which updates $lb = \textsc{cost}(\mathcal{F}, hs) = 0$. As $ub = \infty \neq 0 = lb$, the algorithm continues to the core extraction step. Assume the `Extract-Cores` subroutine returns $K = \{\{b_1, b_2\}, \{b_3, b_4\}\}$ and the solution $\tau = \{b_1, b_2, b_3, b_4, x, \neg y, \neg z\}$. The algorithm then updates $ub = \textsc{cost}(\mathcal{F}, \tau) = 5$. As $lb = 0 < 5 = ub$ the set K is added to $\mathcal{C}$ and the algorithm reiterates. In the next iteration $\mathcal{C} = \{\{b_1, b_2\}, \{b_3, b_4\}\}$ so `Min-Hs` computes (for example) the hitting set $hs = \{b_1, b_3\}$. The lower bound lb is then updated to $\textsc{cost}(\mathcal{F}, hs) = 2 < 5 = ub$ before invoking the next core extraction step. This time around, the first SAT solver call in `Extract-Cores` is done with

the (solver) assumptions $\{\neg b \mid b \in \mathcal{F}_L \setminus hs\} = \{\neg b_2, \neg b_4\}$. The result is SAT, the solver returns the solution $\tau = \{b_1, b_3, \neg b_2, \neg b_4, \neg x, \neg z, y\}$. The procedure `Extract-Cores` then terminates, after which `IHS` updates $ub = \textsc{cost}(\mathcal{F}, \tau) = 2$. Since $ub = lb$, the algorithm terminates and returns τ as an optimal solution of $\mathcal{F}$.

Abstract Cores

The technique of abstract cores is a recently-proposed improvement to `IHS` [6]. An abstract core is a compact representation of a large—potentially exponential—number of regular cores. An abstraction set $ab \subset \mathcal{F}_L$ is a subset of n soft literals that are augmented with count variables $ab.c[1] \dots ab.c[n]$. Informally speaking, the count variables count the number of variables in ab set to true. More precisely, the *definition* of the count variable $ab.c[k]$ is the constraint $ab.c[i] \leftrightarrow \sum_{b \in ab} b \geq i$. An abstract core of an instance $\mathcal{F}$ w.r.t. a collection AB of abstraction sets is then a clause κ that (i) contains only soft literals or count variables and (ii) is entailed by the conjunction of hard clauses of $\mathcal{F}$ and the definitions of count variables.

An IHS algorithm using abstract cores, `IHS-abscores`, extracts both abstract and regular cores during search. Additionally it maintains and dynamically updates a collection AB of abstraction sets over which the abstract cores are then extracted. The abstraction sets are computed based on a graph G that initially has the soft literals as nodes and an edge between any two literals with the same weight that have been found in a core together. The weight of each edge in G between the nodes n_1 and n_2 is the number of times that the literals corresponding to n_1 and n_2 have appeared in cores together. The abstraction sets are then computed by clustering G and using the clusters as abstraction sets. The intuition here is that we wish two literals that often appear in cores together (and are as such in some sense related) to be included in the same abstraction set. During search the quality of the abstraction sets in AB is monitored. If the extracted (abstract) cores are not driving up the lb computed by `Min-Hs`$(\mathcal{F}_L, \mathcal{C})$, then the graph G is reclustered by merging the nodes in the current clusters and then re-clustering the graph.

5.2 Implicit Hitting Sets for Solving MaxSAT under Assumptions

Algorithm 2 details `IHS-assumptions`, our extension of the IHS algorithm to incremental MaxSAT solving under assumptions. The main differences to `IHS` (Algorithm 1) are highlighted in blue. We note that a straightforward extension of `IHS` to MaxSAT solving under assumptions would be to add the clauses $\{(l) \mid l \in A\}$ to the hard clauses of the instance. Doing so would, however, prevent incremental invocations as the SAT solver would need to be reset when removing assumptions between iterations.

Our approach for circumventing the need to reset the SAT solver is based on a concept we call conditional cores. A conditional core of a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$ w.r.t. a set A of assumptions is a clause κ^a containing soft literals and negated assumptions (i.e., $\kappa^a \subset \neg A \cup \mathcal{F}_L$) that is entailed by $\mathcal{F}_H$. The restriction $\textsc{rest}(\kappa^a) = \kappa^a \setminus \neg A$ of a conditional core κ^a is obtained by removing the negated assumptions from it.

Given a MaxSAT instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$ and a set A of assumptions, `IHS-assumptions` maintains a set $\mathcal{C}$ containing the restriction of conditonal cores of $\mathcal{F}$ w.r.t. A. Similarly as `IHS`, `IHS-assumptions` alternates between hitting set computation and core extraction steps. During the hitting set computation steps, a minimum-cost hitting set hs is computed over the restriction of conditional cores in $\mathcal{C}$ exactly as in `IHS` (notice that each such restriction is a subset of $\mathcal{F}_L$). The core extraction steps `Extract-Cores-Assumptions` then extract more cores by iteratively invoking a SAT solver on the clauses of $\mathcal{F}_H$ with

1 `IHS-assumptions`$(\mathcal{F}, \textit{cond-}\mathcal{C}, A)$
  Input: An instance $\mathcal{F} = (\mathcal{F}_H, \mathcal{F}_L, w)$, a set A of assumptions, a set `cond-`$\mathcal{C}$ of conditional cores
  Output: An optimal solution τ under the assumptions A
2 $lb \leftarrow 0$; $ub \leftarrow \infty$; $\tau_{best} \leftarrow \varnothing$; $\mathcal{C} \leftarrow \emptyset$;
3 **for** $\kappa^a \in \textit{cond-}\mathcal{C}$ **do**
4 $\kappa \leftarrow \kappa^a \setminus \neg A$;
5 **if** $\kappa^a \cap A = \emptyset \wedge (\kappa \subset \mathcal{F}_L)$ **then** $\mathcal{C} \leftarrow \mathcal{C} \cup \{\kappa\}$;
6 **while** *(TRUE)* **do**
7 $hs \leftarrow$ `Min-Hs`$(\mathcal{F}_L, \mathcal{C})$;
8 $lb = \textsc{cost}(\mathcal{F}, hs)$;
9 **if** $(lb = ub)$ **then break**;
10 $(K, \tau) \leftarrow$ `Extract-Cores-Assumptions`$(\mathcal{F}_H, \mathcal{F}_L, A, hs)$;
11 **if** $(\textsc{cost}(\mathcal{F}, \tau) < ub)$ **then** $\tau_{best} \leftarrow \tau$; $ub \leftarrow \textsc{cost}(\mathcal{F}, \tau)$;
12 **if** $(lb = ub)$ **then return** τ_{best};
13 **for** $\kappa^a \in K$ **do**
14 `cond-`$\mathcal{C} \leftarrow$ `cond-`$\mathcal{C} \cup \{\kappa^a\}$;
15 $\mathcal{C} \leftarrow \mathcal{C} \cup \{\kappa^a \setminus \neg A\}$;

Algorithm 2 Incremental MaxSAT solving under assumptions

`Solver-A` $= (\{\neg b \mid b \in \mathcal{F}_L \setminus hs\} \setminus \{l \mid l \in \mathcal{F}_L \cap \neg A\}) \cup A$ as the (solver) assumptions. If the result is "satisfiable", the solver returns a model $\tau \supset A$ which is feasible for the MaxSAT instance under the set of assumptions A. If the result is "unsatisfiable", the solver returns a subset $\kappa \subset \neg$`Solver-A`, i.e., a conditional core. At the end of each iteration the restriction $\textsc{rest}(\kappa^a)$ of each κ^a computed during `Extract-Cores-Assumptions` is added to $\mathcal{C}$. Additionally, all conditional cores are added to a set `cond-`$\mathcal{C}$ to be stored in between iterations.

The correctness of `IHS-assumptions` follows from the fact that $\mathcal{F}_H$ entails a conditional core κ^a if and only if $\mathcal{F}_H \wedge \bigwedge_{l \in A}(l)$ entails its restriction $\textsc{rest}(\kappa^a)$. In other words, every conditional core of $\mathcal{F}$ w.r.t. the assumptions A is a (standard) core of the instance $(\mathcal{F}_H \wedge A, \mathcal{F}_L, w)$. In fact, given a clause C entailed by $\mathcal{F}_H$ and a set A of assumptions, C is a conditional core w.r.t. A if and only if $(C \setminus \neg A) \subset \mathcal{F}_L$. This allows for initializing the set $\mathcal{C}$ by the restrictions of conditional cores computed in previous iterations that are also conditional cores w.r.t. the current set of assumptions (Lines 3-5 of Algorithm 2).

The concept of conditional cores extends to abstract cores: an abstract conditional core w.r.t. a set of assumptions A is a clause that (i) contains soft literals, count variables or negations of literals in A and (ii) is entailed by $\mathcal{F}_H$ and the definitions of the count variables.

▶ **Example 4.** Consider an invocation of `IHS-assumptions` on the MaxSAT instance $\mathcal{F}$ from Example 1 under the assumptions $A = \{x\}$. For clarity, here we will ignore the set `cond-`$\mathcal{C}$. Initially $\mathcal{C} = \emptyset$, so the first hitting set $hs = \emptyset$, As such $lb = \textsc{cost}(\mathcal{F}, hs) = 0 < \infty = ub$, so the algorithm invokes `Extract-Cores-Assumptions`$(\mathcal{F}_H, \mathcal{F}_L, \{x\}, \emptyset)$. The procedure extracts conditional cores of $\mathcal{F}$ by invoking a SAT solver under the assumptions `Solver-A` $= \{\neg b \mid b \in \mathcal{F}_L \setminus hs\} \cup A = \{\neg b_1, \neg b_2, \neg b_3, \neg b_4, x\}$. The result is "unsatisfiable". Assume the solver returns the conditional core $\kappa^a = \{\neg x, b_2\}$. The next solver call is made under the assumptions `Solver-A` $= \{\neg b_1, \neg b_3, \neg b_4, x\}$. The result is again "unsatisfiable" and the solver returns the (conditional) core $\{b_3, b_4\}$. The third solver call returns "satisfiable" and (for example) the solution $\tau = \{\neg b_1, b_2, \neg b_3, b_4, x, y, \neg z\}$ so `Extract-Cores-Assumptions`

terminates. The upper bound is updated by $ub = \text{COST}(\mathcal{F}, \tau) = 3$ and the restrictions of each conditional core added to $\mathcal{C}$. In the next iteration $\mathcal{C} = \{\{b_2\}, \{b_3, b_4\}\}$ and the `Min-Hs` procedure returns (for example) $hs = \{b_2, b_3\}$. This hitting set updates the lower bound to $lb = \text{COST}(\mathcal{F}, hs) = 3 = ub$ so the algorithm terminates and returns τ as an optimal solution to $\mathcal{F}$ under A.

5.3 Implementing IPAMIR Functionality in the MaxHS Solver

With the necessary concepts in place, we now describe our implementation of the IPAMIR functions within the state-of-the-art IHS MaxSAT solver MaxHS [8, 6], resulting in a truly incremental MaxSAT solver. We extend further on our previous extension of MaxHS which was focused on enabling incremental computations w.r.t. CHANGEWEIGHT (partially covering `ipamir_add_soft_lit`), described in [27]. It should be noted that the current version of MaxHS goes well beyond this basic description of IHS, employing a variety of runtime improving techniques [3, 6, 9] and implementation-level "tricks". As we develop a concrete incremental extension of MaxHS, it is important to consider in more detail some of these techniques—in particular, some of these techniques can result in incorrectness if not carefully taken into account in an incremental extension of MaxHS.

Managing Variables

MaxHS implements a class for managing different variables depending on their type: original, soft, count (for implementing abstract cores). We extend this to allow for setting an existing variable to a soft literal (to allow for using `ipamir_add_soft_lit` after solving) and creating new variables (to allow for using `ipamir_add_hard` and `ipamir_assume` after solving). We also extend variable types to cover user-provided assumptions. This way we can straightforwardly check whether a literal in a core reported by the SAT solver is a user-provided assumption, never removing it from the assumptions given to the SAT solver. Similarly, restrictions of conditional cores are easily computed this way.

Internal SAT Solver and MUS Extractor

We extend the internal SAT solver to allow for performing unit propagation on the user-provided assumption literals. This allows for sharing the derived soft literals to solvers computing minimum hitting sets, namely CPLEX as an IP solver and a greedy hitting set solver. In addition to fixed truth values the SAT solver derives from the formula, user-provided assumption literals and ones derived from them via unit propagation are also considered fixed values during an iteration. In MaxHS these fixed values are used for, e.g., removing redundant assumptions from SAT solver calls and computing an initial lower bound. Extracted conditional cores containing assumptions can be soundly minimized using the internal minimal unsatisfiable core extractor. However, we do not immediately return a unit conflict if a literal in the core has been fixed due to assumptions, unlike if it has been fixed otherwise.

Reduced Cost Fixing

MaxHS determines whether a soft literal can be fixed using so-called reduced costs extracted from the optimal solution of the LP relaxation of the hitting set problem [3]. As noted in [27], this is incorrect under changes to the weights in a MaxSAT instance. Similarly, due to supporting user-provided assumptions via `ipamir_assume`, the opposite of a literal that

has been fixed via reduced costs could be assumed, leading to an empty conflict. However, the fixing of variables based on the reduced costs of the current hitting set problem can be implemented correctly using assumptions. In particular, if a soft literal is fixed due to reduced costs, instead of adding the corresponding unit clause to the SAT solver, we add it to the set of user-provided assumptions. As assumptions are cleared after each solve call, reduced cost fixings become reverted between iterations.

Abstract Cores

The abstraction sets employed for abstract cores in the incremental MaxHS are computed on a graph structure whose nodes correspond to the soft literals of the instance. The weight of an edge between two nodes n_1 and n_2 indicates the number of times a conditional core containing both a literal corresponding to n_1 and a literal corresponding to n_2 has been extracted. When building an abstraction set, MaxHS attempts to compute a lower bound $k \geq 1$ on the number of count variables assigned to 1 by any solutions of the instance. (This resembles the so called core-exhaustion technique employed by core-guided MaxSAT solvers [16].) In the incremental version, we perform this step independently of the current assumptions. In other words, we compute lower bounds on abstraction sets that hold for any solution of the instance, not only solutions that satisfy the current assumptions. As a consequence, any assignments obtained during this lower-bounding phase need not satisfy the assumptions and as such might not be feasible solutions to the current iteration. Even so, the intuition here is that the benefits of computing lower bounds usable also in future iterations outweigh the drawbacks of not obtaining feasible solutions for the current instance. Finally, as detailed in [27], changing the weights of any soft literals invalidates any abstraction sets that now contain literals with differing weights. We allow the soft literals in any invalidated abstraction sets to be reintroduced into the graph in order to possibly be assigned to future abstraction sets.

Conditional Core Database

To store conditional cores, we use an additional SAT solver instance to which all conditional cores are added as clauses. When extracting a set of cores to be added to the IP solver, we perform unit propagation using current user-provided assumptions. Afterwards we check each clause remaining in the database. Each clause that at this point only contains soft literals or count variables is added to the IP solver. The intuition here is that unit propagation simulates the computation of the relaxation of each conditional core in the database wrt the current set of assumptions. Note however that using unit propagation rather than a straight-forward application of the definitions (as is done on Lines 3-5 of Algorithm 2) can allow for the computation of more and smaller valid cores.

▶ **Example 5.** Let $A = \{x, \neg b_2\}$ be the set of assumptions to be propagated on the database containing the clauses $\{\{\neg x, b_1\}, \{\neg x, y, b_2\}, \{\neg b_1, b_3, b_4\}, \{\neg y, b_5, b_6\}\}$ that represent conditional cores found in earlier iterations. Assume that each b_i for $i = 1, \ldots, 4$ is a soft literal and x and y are other (non count) literals. After running unit propagation to fixpoint, the database contains the clauses $\{\{b_1\}, \{y\}, \{b_3, b_4\}, \{b_5, b_6\}\}$. From these the clauses $\{b_1\}, \{b_3, b_4\}$ and $\{b_5, b_6\}$ are obtained as valid cores to add to the IP solver. In contrast, checking if $\textsc{rest}(C) = C \setminus \neg A$ contains only soft literals for each clause in the database would give the cores $\{b_1\}$ and $\{\neg b_1, b_3, b_4\}$.

Each valid core obtained after unit propagation is added to the IP solver and the greedy hitting set solver whenever they are reinitialised, which in turn happens at the start of

each iteration prior to which new soft literals have been declared or assumptions have been provided.

IPAMIR Wrapper

Before `ipamir_solve` is called, `ipamir_add_hard` and `ipamir_add_soft_lit` operate on the internal WCNF data structure. When `ipamir_solve` is called for the first time, simplification on the WCNF is invoked and the MaxSAT solver is initialized. After this, implementing the IPAMIR functions requires more attention. Since `ipamir_add_hard`, `ipamir_add_soft_lit`, and `ipamir_assume` make use of external literals, we map them to internal ones, making sure that literals which have been fixed by simplification are handled correctly. That is, if the literal has been fixed to true, we skip the clause, soft literal, or the assumption; if to false, we skip it from the clause (for `ipamir_add_hard`), increase the base cost (for `ipamir_add_soft_lit`), or set the UNSAT flag to true (for `ipamir_assume`). Now, the internal hard clause added using `ipamir_add_hard` can be directly added to the internal SAT solver. If the added hard clause is falsified under the current model, the upper bound is set to ∞. The lower bound is set to 0.

To add a new soft literal via `ipamir_add_soft_lit`, no changes to the SAT solver are necessary. Instead, the literal is declared as a soft literal in the corresponding data structures. However, if the user calls `ipamir_add_soft_lit` with a negative literal $\neg x$ after `ipamir_solve` has been called, since MaxHS does not support negative soft literals, we instead add a binary hard clause $b \vee \neg x$ with a fresh variable b, which is declared soft. If `ipamir_add_soft_lit` is called with a soft literal that has already been declared, we instead make use of the weight changing procedure implemented previously [27].

Finally, user-provided assumptions added via `ipamir_assume` are stored in a separate container which the main solving procedure has access to.

WCNF Simplification

Finally, we note that MaxHS employs by default several simplification techniques which are applied to the input formula before calling the solving procedure. We overview the effect of these simplification techniques on the implementation of incremental functions.

- *Weight-based hardening of soft clauses.* MaxHS attempts to harden soft clauses based on their (high) weight. This is not directly applicable in incremental MaxSAT solving. For example, a hardened unit soft clause added via `ipamir_add_soft_lit` can be assumed false using `ipamir_assume`. This is also the case for changing weights [27]. Similarly to [27], we disable this functionality in the simplification procedure. In contrast, at the beginning of each `solve` call, we check which soft clauses can be hardened given the current assumptions and weights. Similarly as for reduced cost fixing, we add the corresponding soft literal forcing a soft clause to be satisfied to the current set of assumption literals.
- *Unit propagation and equality detection.* MaxHS performs two rounds of unit propagation (on the hard clauses) and equality detection (on the hard and soft clauses), and eliminates pure literals between these rounds. Pure literal elimination is not directly applicable for incremental solving: for example, the negation of the literal can be assumed via `ipamir_assume` or used in a hard clause via `ipamir_add_hard`. Thus, this feature is disabled. To allow for using literals which have been eliminated due to unit propagation, we remember which units have been derived. Similarly, to allow for using literals which have been replaced by their equal representative, we remember which equalities have been detected.

Finally, variables are remapped from external variables to internal variables. We make use of these mappings when adding hard clauses, soft literals and assumptions. Additional techniques have been discussed from the perspective of changing weights in [27].

6 Empirical Evaluation

We turn to an empirical evaluation comparing the runtime efficiency of our incremental MaxHS extension called iMaxHS (available in open source) to that of the the MaxSAT Evaluation 2021 version of MaxHS on which our incremental extension is based on. For the experiments, the original non-incremental MaxHS was run in default settings. The experiments were run on machines with 8-core 2.60-GHz Intel Xeon E5-2670 CPUs and 57-GB RAM under Red Hat Enterprise Linux 8.5. A per-instance timeout limit of 7200 seconds (2 hours) and a memory limit of 16 GB was enforced. Specifically, the 7200-second time limit is for solving a specific MaxSAT instance n times with n different assumptions $\mathcal{A}_1, \ldots, \mathcal{A}_n$. For both the incremental and non-incremental solver, we record the solving time t_k of each iteration k. The kth iteration as well as all subsequent ones are considered timed out if $\sum_{i=1}^{k} t_i > 7200$ seconds.

6.1 Benchmarks

As benchmarks we used all 1184 instances from the complete tracks of MaxSAT Evaluation 2021, including both weighted and unweighted instances and covering a wide range of different (non-random) problem domains. For each MaxSAT Evaluation 2021 instance, we construct an incremental benchmark, constituting of 20 MaxSAT solver calls, each under a different set of assumptions. Each set of assumptions is obtained by hardening each soft clause uniformly at random with probability $p = 1/100$. All in all, starting from the 1184 MaxSAT Evaluation instances, this results in a total of 23680 iterations overall. As the original non-incremental MaxHS does not support assumptions, we instead add the hardened soft clauses directly as hard clauses for each iteration of a benchmark. To focus the comparison on actual runtime effects rather than potentially over-emphasizing parsing times of the original MaxHS, we exclude the WCNF parsing times from the CPU times when reporting the results.

6.2 Results

In the default version of iMaxHS, we initialize the set of conditional cores by solving the original instance (without assumptions) for a maximum of 100 seconds (for the release version, this time limit is a user-controlled option). In addition, we perform the initial disjoint phase—where an initial set of disjoint cores are extracted before calling the hitting set solver—only once per benchmark. If this initial phase is completed during initialization, we do not perform it again during solver calls for assumptions. To investigate the impact of different parameter choices in iMaxHS in addition to comparing its runtime performance in its default settings to the performance of the original non-incremental MaxHS, we consider three further configurations of iMaxHS.

- iMaxHS-NOI: does not perform the initialization phase (where the original instance is solved without assumptions).
- iMaxHS-NOC: does not make use of conditional cores, instead using only cores found during initialization.
- iMaxHS-DJA: performs the initial disjoint phase at each iteration.

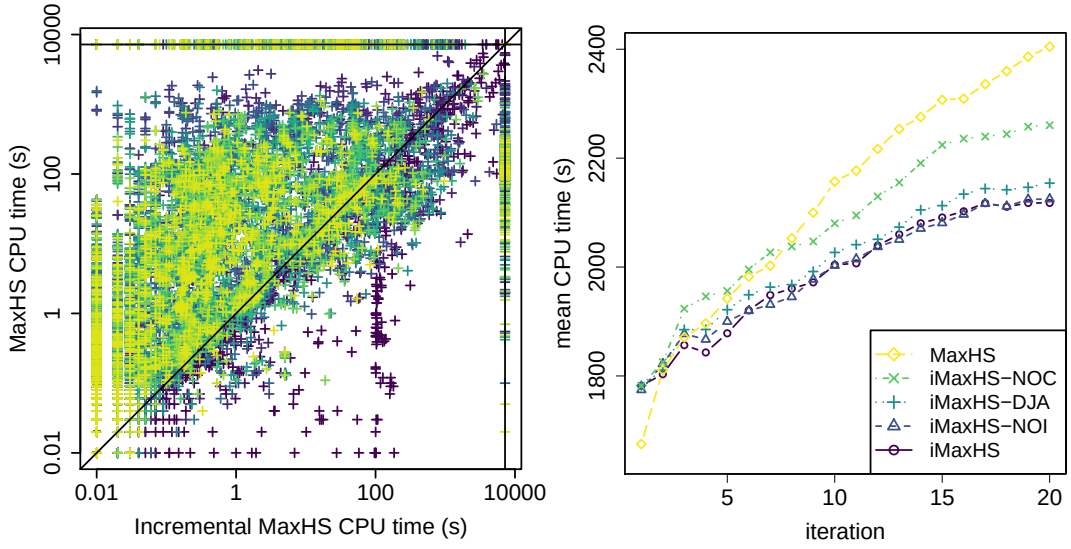

Figure 4 Left: Per-iteration runtime comparison of iMaxHS and MaxHS; blue points correspond to earlier iterations and yellow points to later ones. Right: Mean runtime per-iteration for MaxHS and different configurations of iMaxHS.

Figure 4 (left) shows a per-iteration runtime comparison between the default configuration of iMaxHS (x-axis) and MaxHS (y-axis). Points are colored by the iteration index: the more blue a point, the earlier the iteration; the more yellow, the later the iteration. We observe that iMaxHS outperforms MaxHS overall, with the exception of the first iteration taking approximately 100 seconds on some instances which MaxHS solves quickly. This is due to the runtime allocation of 100 seconds for the initialization phase in iMaxHS. Overall, there are 6450 timeouts for iMaxHS against 6821 timeouts for MaxHS. Figure 4 (right) shows a runtime comparison of different configurations of iMaxHS and MaxHS in terms of mean solving time per each iteration over all benchmarks. Timeouts are included as the timeout limit (7200 seconds). We observe that already after 2 iterations, incrementality of iMaxHS starts to pay off. The gap between iMaxHS and the non-incremental MaxHS increases significantly with the number of iterations for the benefit of iMaxHS. Interestingly, comparing iMaxHS and iMaxHS-DJA, it appears that performing the initial disjoint phase is somewhat detrimental to the performance. This may indicate that cores extracted during the initial disjoint phase are somewhat similar to the later extracted cores, resulting in an unnecessary runtime overhead for iMaxHS-DJA. The iMaxHS-NOC configuration exhibits weaker performance when compared to the other incremental configurations. This suggests that conditional cores gathered during earlier iterations are essential for making use of incrementality.

Finally, Figure 5 shows the number of solved instances at iterations $k = 2, 5, 10$ for different per-instance time limits. The benefits of incrementality are clear already at the second iteration (Figure 5 left): all incremental versions—especially the default iMaxHS—exhibit faster runtimes than MaxHS on average. Comparing iMaxHS to iMaxHS-NOI, the benefit of initialization is clear at $k = 2$, as the default iMaxHS is faster on average. We also observe that iMaxHS-NOC is on average slower than all other configurations especially on the harder instances. The benefits of incrementality become increasingly clear as more iterations are made. On the fifth iteration (Figure 5 center) iMaxHS is significantly faster than MaxHS. We observe that iMaxHS-DJA is significantly slower than iMaxHS, again confirming that

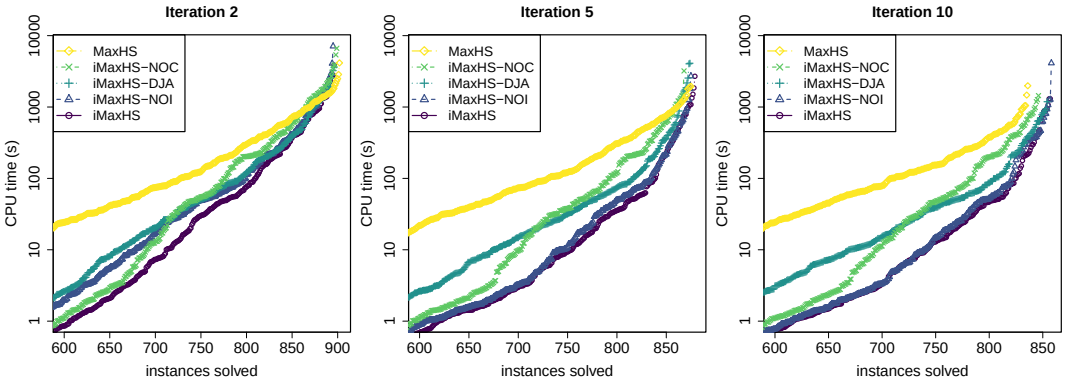

Figure 5 Cactus plots (number of solved instances for a CPU time limit) for different iterations.

performing the initial disjoint phase only once is beneficial for overall performance. iMaxHS-NOC is clearly not competitive with other versions of iMaxHS, emphasizing the importance of conditional cores. Finally, the picture remains similar when comparing the fifth and tenth iterations (Figure 5 right).

7 Related Work

There is some earlier work on enabling incrementality in MaxSAT solving, although in much more restricted ways as what we develop in this work. In [32] the authors investigate a restricted form of incremental MaxSAT that only allows adding hard and soft clauses in the context of core-guided MaxSAT, in particular for the classical Fu-Malik algorithm [14]. Since adding clauses does not invalidate any cores, the algorithm can preserve all of the found cores between iterations. The authors suggest to periodically restart the search—thus removing all discovered cores—to improve performance. We believe that it would be non-trivial to extend the algorithm of [32] to support the full IPAMIR interface (i.e., assumptions and changing weights).

A lazy grounding framework for solving large MaxSAT instances was presented in [21]. The idea is to solve a large instance $\mathcal{F}$ by solving a sequence of instances consisting of different subsets of the clauses in $\mathcal{F}$. The paper proposes a variety of ways of selecting which clauses to lazily include in the next instance of the sequence. To the best of our understanding, however, the implementation experimented with does not make use of any forms of incrementality on the MaxSAT-level but instead resets the MaxSAT solver between each iteration.

Finally, we note that so-called incremental cardinality constraints commonly applied in modern core-guided MaxSAT solvers [25, 24, 23] refer to incrementality on the *SAT-level*, allowing the core transformations in the core-guided approach to be performed without reseting the internal SAT solver. This is different from enabling incremental computations on the *MaxSAT-level*, i.e., incrementally solving a sequence of related MaxSAT instances under various types of changes.

8 Conclusions

Incremental computations are today widely supported by SAT solvers, and are a key to the employment of SAT solvers as practical NP oracles in various applications. In analogy, enabling truly incremental computations for solving sequences of related optimization problem

instances has the potential for major performance improvements. In this work, we made several contributions towards incremental MaxSAT solving. Building on the IPASIR interface from the realm of SAT, we proposed the IPAMIR interface as a standard for supporting incremental functionality in MaxSAT solvers and for developing applications of incremental MaxSAT solving. Extending the successful implicit hitting set approach to MaxSAT solving as a prime candidate for enabling incremental computations, we provided what we believe to be the first openly-available incremental MaxSAT solver in its generality. In particular, iMaxHS supports the full range of functionality specified by IPAMIR. Complementing recent empirical evidence on the benefits of incremental computations under changes to soft clause weights, we provided empirical evidence on the potential of incrementally solving the same MaxSAT instances under varying sets of assumptions.

In terms of further work, in the specific context of IHS-based MaxSAT solving there is potential for improving the performance of our current incremental extension of MaxHS by a more in-depth study of e.g. different heuristic choices and implementation-level aspects. More generally, we hope and believe IPAMIR offers a basis for developing truly incremental functionality into MaxSAT solvers more widely, with a promising line of development in further extending the reach of MaxSAT as a paradigm of choice to solving combinatorial optimization problems. This also opens up various non-trivial research questions, such as the challenge of enabling strong preprocessing and other forms of extended reasoning techniques in incremental MaxSAT solving.

References

1 Mario Alviano, Carmine Dodaro, and Francesco Ricca. A MaxSAT algorithm using cardinality constraints of bounded size. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2677–2683. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/379.

2 Carlos Ansótegui, Frédéric Didier, and Joel Gabàs. Exploiting the structure of unsatisfiable cores in MaxSAT. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 283–289. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/046.

3 Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 641–651. Springer, 2017. doi:10.1007/978-3-319-66158-2_41.

4 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability, Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

5 Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT Race 2015. *Artificial Intelligence*, 241:45–65, 2016. doi:10.1016/j.artint.2016.08.007.

6 Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.

7 Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. In Matti Järvisalo and Allen Van Gelder,

editors, *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013, Proceedings*, volume 7962 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2013. doi:10.1007/978-3-642-39071-5_12.

8 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011, Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.

9 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013, Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013. doi:10.1007/978-3-642-40627-0_21.

10 Emir Demirovic. SAT-based approaches for the general high school timetabling problem. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 5175–5176. ijcai.org, 2017. doi:10.24963/ijcai.2017/747.

11 Bruno Dutertre. An empirical evaluation of SAT solvers on bit-vector problems. In François Bobot and Tjark Weber, editors, *Proceedings of the 18th International Workshop on Satisfiability Modulo Theories co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Online (initially located in Paris, France), July 5-6, 2020*, volume 2854 of *CEUR Workshop Proceedings*, pages 15–25. CEUR-WS.org, 2020. URL: http://ceur-ws.org/Vol-2854/paper1.pdf.

12 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003. doi:10.1016/S1571-0661(05)82542-3.

13 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019. doi:10.1007/978-3-030-24258-9_9.

14 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006. doi:10.1007/11814948_25.

15 Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with MaxSAT and its integration in AdaBoost. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1170–1176. ijcai.org, 2020. doi:10.24963/ijcai.2020/163.

16 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019. doi:10.3233/SAT190116.

17 Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. Minimal model generation with respect to an atom set. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 7th International Workshop on First-Order Theorem Proving, FTP 2009, Oslo, Norway, July 6-7, 2009*, volume 556 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009. URL: http://ceur-ws.org/Vol-556/paper06.pdf.

18 Alexandre Lemos, Pedro T. Monteiro, and Inês Lynce. Minimal perturbation in university timetabling with maximum satisfiability. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020,*

Proceedings, volume 12296 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 2020. doi:10.1007/978-3-030-58942-4_21.

19 Florian Lonsing and Uwe Egly. Evaluating QBF solvers: Quantifier alternations matter. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 276–294. Springer, 2018. doi:10.1007/978-3-319-98334-9_19.

20 Ravi Mangal, Xin Zhang, Aditya Kamath, Aditya V. Nori, and Mayur Naik. Scaling relational inference using proofs and refutations. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3278–3286. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12466.

21 Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. Volt: A lazy grounding framework for solving very large MaxSAT instances. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 299–306. Springer, 2015. doi:10.1007/978-3-319-24318-4_22.

22 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 133 – 182. IOS Press, 2021. doi:10.3233/FAIA200987.

23 Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014, Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2014. doi:10.1007/978-3-319-10428-7_39.

24 Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. On using incremental encodings in unsatisfiability-based MaxSAT solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):59–81, 2014. doi:10.3233/sat190102.

25 Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014, Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 438–445. Springer, 2014. doi:10.1007/978-3-319-09284-3_33.

26 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2717–2723. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8513.

27 Andreas Niskanen, Jeremias Berg, and Matti Järvisalo. Enabling incrementality in the implicit hitting set approach to MaxSAT under changing weights. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 44:1–44:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.44.

28 Andreas Niskanen and Matti Järvisalo. Strong refinements for hard problems in argumentation dynamics. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 841–848. IOS Press, 2020. doi:10.3233/FAIA200174.

29 Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2018. doi:10.1007/978-3-319-94144-8_3.

30 Matthew Richardson and Pedro M. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006. doi:10.1007/s10994-006-5833-1.

31 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 539–546. Springer, 2016. doi:10.1007/978-3-319-40970-2_34.

32 Xujie Si, Xin Zhang, Vasco M. Manquinho, Mikolás Janota, Alexey Ignatiev, and Mayur Naik. On incremental core-guided MaxSAT solving. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 473–482. Springer, 2016. doi:10.1007/978-3-319-44953-1_30.

33 Xin Zhang, Ravi Mangal, Aditya V. Nori, and Mayur Naik. Query-guided maximum satisfiability. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 109–122. ACM, 2016. doi:10.1145/2837614.2837658.