Enabling Incrementality in the Implicit Hitting Set Approach to MaxSAT under Changing Weights

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

Recent advances in solvers for the Boolean satisfiability (SAT) based optimization paradigm of maximum satisfiability (MaxSAT) have turned MaxSAT into a viable approach to finding provably optimal solutions for various types of hard optimization problems. In various types of real-world problem settings, a sequence of related optimization problems need to solved. This calls for studying ways of enabling incremental computations in MaxSAT, with the hope of speeding up the overall computation times. However, current state-of-the-art MaxSAT solvers offer no or limited forms of incrementality. In this work, we study ways of enabling incremental computations in the context of the implicit hitting set (IHS) approach to MaxSAT solving, as both one of the key MaxSAT solving approaches today and a relatively well-suited candidate for extending to incremental computations. In particular, motivated by several recent applications of MaxSAT in the context of interpretability in machine learning calling for this type of incrementality, we focus on enabling incrementality in IHS under changes to the objective function coefficients (i.e., to the weights of soft clauses). To this end, we explain to what extent different search techniques applied in IHS-based MaxSAT solving can and cannot be adapted to this incremental setting. As practical result, we develop an incremental version of an IHS MaxSAT solver, and show it provides significant runtime improvements in recent application settings which can benefit from incrementality but in which MaxSAT solvers have so-far been applied only non-incrementally, i.e., by calling a MaxSAT solver from scratch after each change to the problem instance at hand.

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

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

Digital Object Identifier 10.4230/LIPIcs.CP.2021.42

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, 328718 and 342145.

1 Introduction

Maximum satisfiability (MaxSAT) constitutes today a viable approach to solving various types of NP-hard real-world optimization problems (see [6] for a recent survey). This is in particular due to various recent algorithmic advances applied in readily-available MaxSAT solvers. Iteratively solving a series of decision problems with Boolean satisfiability solvers gives a basis for most if not all current state-of-the-art MaxSAT solvers [4, 5]. MaxSAT solvers make heavy use of the incremental APIs of SAT solvers [13], through which SAT solvers can provide explanations as unsatisfiable subsets of soft constraints (i.e., unsatisfiable cores). Two main paradigms adhering to this framework are the core-guided approach (see e.g. [24, 23, 25, 1, 2, 17]) and the implicit hitting set (IHS) approach [11, 12, 10, 29]. In the core-guided approach, cores iteratively obtained from the SAT solvers are used for transforming the original MaxSAT instance in a controlled way so that, once satisfiability is achieved, any satisfying truth assignment reported by the SAT solver constitutes an optimal solution to the original MaxSAT instance. The IHS approach, on the other hand, leaves the original MaxSAT instance unchanged, and computes at each iteration a hitting set of the so-far accumulated set of cores. In the next SAT solver call, the soft clauses contained in the most recently computed hitting set are ignored. This loop is continued essentially (i.e., ignoring various search techniques applied in actual solver implementations of the IHS approach) until the SAT solver returns a satisfying truth assignment.

Both of these SAT-based MaxSAT solving paradigms make heavy use of incremental computations on the level of the SAT solver. However, enabling incremental computations on the actual MaxSAT solving level, i.e., gearing MaxSAT solvers towards solving sequences of related MaxSAT instances without restarting search for each instance, remains today an underdeveloped research direction. Indeed, MaxSAT solvers offer little for such incremental settings, with the exception of a few solver implementations offering an API for adding hard and soft clauses in-between the MaxSAT solver calls [29, 17]. This kind of incremental solving has been further investigated in the context of core-guided solving by adaptively restarting the solver when the quality of the cores degrades [30]. Note that, while so-called incremental cardinality constraints have been proposed and are applied in core-guided MaxSAT solvers [22, 21, 20], this notion refers to incrementality on the SAT-level within the core transformations in the core-guided approach rather than incrementality on the level of MaxSAT, i.e., in incrementally solving a sequence of MaxSAT instances. The lack of support for more generic forms of incrementality on the level of MaxSAT is indeed problematic: various types of recent real-world applications of MaxSAT solvers [19, 9, 33, 27] could evidently benefit in terms of runtime improvements with the help of incremental computations, but currently have to resort to calling a MaxSAT solver from scratch for each instance that needs to be solved towards finding an optimal solution to the problem at hand.

In this work, we make progress on enabling incremental computations on the MaxSAT level. Specifically, we focus on enabling incrementality in problem settings constituting of solving a sequence of MaxSAT instances which differ from each other in the weights of the soft clauses in the instances. In particular, we consider the general setting where the soft clause weights of the next instance in the sequence are adaptively assigned based on the previous instances in the sequence and the optimal solutions found to those instances. Interestingly, this form of incrementality can be identified to be intrinsically present in various application settings of MaxSAT in the context of interpretable machine learning [18, 15, 16, 32] (though various other types of application settings can naturally be imagined), but is not supported by any of the state-of-the-art MaxSAT solvers.

Specifically, we focus on enabling incrementality under changing soft clause weights in the context of the IHS approach to MaxSAT solving. The IHS approach is particularly appealing for incrementality due to the very fact that the solving process does not essentially alter (through core transformations, as in the core-guided approach) the original MaxSAT instance. This allows for ensuring that cores between different MaxSAT instances under changing weights can be reused across the different instances and hence need not be re-computed from scratch. However, the various intricate search techniques and optimizations implemented in the state-of-the-art IHS MaxSAT solver MaxHS make adapting the solver for incremental computations under changing weights a non-trivial task in practice. To this end, we describe in detail the search techniques that can and cannot be applied in incremental computations under changing soft clause weights, and provide a first IHS solver implementation supporting incrementality under changing weights, building on MaxHS. Most concretely, we apply this incremental adaptation of MaxHS to two recent applications of MaxSAT solving in the context of interpretable machine learning, namely, decision tree boosting and decision set learning, identifying that both of these problem settings could at least in principle benefit in terms of overall runtimes of incremental computations under changing weights. Indeed, we show that our adaptation of MaxHS supporting incrementality under changing weights provides significant runtime improvements compared to a current version of (non-incremental) MaxSAT, despite the fact that not all performance-optimizing techniques applied in the non-incremental version can be applied by the incremental adaptation.

2 Maximum Satisfiability

For a Boolean variable x, there are two literals x and $\neg x$. A clause C is disjunction of literals, viewed as a set, and a (CNF) formula F is a conjunction of clauses, again viewed as a set. A truth assignment τ maps Boolean variables to 1 (true) or 0 (false). The semantics of truth assignments are extended 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 assignment τ is a model of a formula F if $\tau(F) = 1$. A formula F is satisfiable if it has a model, otherwise it is unsatisfiable.

An instance $\mathcal{F}$ of (weighted partial) MaxSAT consists of two CNF formulas, the hard clauses $\mathrm{H}(\mathcal{F})$ and the soft clauses $\mathrm{S}(\mathcal{F})$, and a weight function $w(\mathcal{F})\colon \mathrm{S}(\mathcal{F}) \to \mathbb{R}^+$ that assigns a positive weight to all soft clauses. When the instance $\mathcal{F}$ is clear from context, we use H, S and w to denote $\mathrm{H}(\mathcal{F})$, $\mathrm{S}(\mathcal{F})$ and $w(\mathcal{F})$, respectively. A model τ of H is a solution to $\mathcal{F}$. We assume that MaxSAT instances have at least one solution, i.e., that H is satisfiable. A solution τ to $\mathcal{F}$ has cost $\textsc{cost}(\mathcal{F}, \tau) = \sum_{C \in \mathrm{S}} w(C)(1 - \tau(C))$, i.e., 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 all solutions τ' of $\mathcal{F}$. We denote the cost of the optimal solutions to $\mathcal{F}$ by $\textsc{cost}(\mathcal{F})$. When convenient, we treat a solution τ as the set of literals the assignment satisfies, i.e, as $\tau = \{l \mid \tau(l) = 1\}$.

In order to simplify notation we will assume that each soft clause $C \in \mathrm{S}$ is a unit soft clause containing a negation of a variable, i.e., $C = (\neg b)$. This assumption can be made without loss of generality as any soft clause $C \in \mathrm{S}$ can be transformed into the hard clause $C \vee b$ and the soft clause $(\neg b)$ with $w((\neg b)) = w(C)$ where b is a new variable. A variable b that appears in a soft clause $(\neg b) \in \mathrm{S}$ is a *blocking* variable; we denote the set of blocking variables of $\mathcal{F}$ by $\mathcal{B}(\mathcal{F})$. As assigning a blocking variable $b = 1$ corresponds to falsifying the corresponding soft clause $(\neg b)$, we treat blocking variables and soft clauses interchangeably, and extend the weight function w to blocking variables by $w(b) = w((\neg b))$ and to sets $B_s \subset \mathcal{B}(\mathcal{F})$ by $\textsc{cost}(\mathcal{F}, B_s) = \sum_{b \in B_s} w(b)$. The cost of a solution τ is then $\textsc{cost}(\mathcal{F}, \tau) = \sum_{b \in \mathcal{B}(\mathcal{F})} \tau(b) w(b)$.

The IHS algorithm for computing optimal MaxSAT solutions, focused on in this work, makes use of so-called (unsatisfiable) cores of MaxSAT instances. A core $\kappa \subset \mathrm{S}$ is a set of soft clauses that is unsatisfiable together with the hard clauses. As each soft clause $C \in \kappa$ is a negation of a variable $C = (\neg b)$, the fact that $\mathrm{H} \wedge \kappa = \mathrm{H} \wedge \bigwedge_{(\neg b) \in \kappa} (\neg b)$ is unsatisfiable implies that any solution to $\mathcal{F}$ assigns $b = 1$ for at least one $(\neg b) \in \kappa$. Thus a core can be expressed as the clause $\{b \mid (\neg b) \in \kappa\}$ that is entailed by H, i.e., a clause over blocking variables that is satisfied by all solutions to $\mathcal{F}$. Note that κ can also be expressed as the

```
1  IHS($\mathcal{F}$)
     Input: An instance $\mathcal{F} = (\mathrm{H}, \mathrm{S}, w)$
     Output: An optimal solution $\tau$
2    $lb \leftarrow 0$; $ub \leftarrow \infty$;
3    $\tau_{best} \leftarrow \varnothing$; $\mathcal{C} \leftarrow \emptyset$;
4    while ($\textsc{true}$) do
5        $hs \leftarrow$ Min-Hs($\mathcal{B}(\mathcal{F}), \mathcal{C}$);
6        $lb = \textsc{cost}(\mathcal{F}, hs)$;
7        if ($lb = ub$) then break;
8        $(K, \tau) \leftarrow$ Extract-Cores($\mathrm{H}, \mathcal{B}(\mathcal{F}), hs$);
9        if ($\textsc{cost}(\mathcal{F}, \tau) < ub$) then
           $\tau_{best} \leftarrow \tau$; $ub \leftarrow \textsc{cost}(\mathcal{F}, \tau)$;
10       if ($lb = ub$) then return $\tau_{best}$;
11       $\mathcal{C} \leftarrow \mathcal{C} \cup K$;
```

Algorithm 1 IHS for MaxSAT

$$
\begin{array}{ll}
\textbf{minimize} & \displaystyle\sum_{b \in \mathcal{B}(\mathcal{F})} w(b) \cdot b \\
\textbf{subject to} & \\
\sum_{b \in \kappa} b \geq 1 & \forall \kappa \in \mathcal{C} \\
b \in \{0, 1\} & \forall b \in \mathcal{B}(\mathcal{F})
\end{array}
$$

Figure 1 An integer program for computing a hitting set over a set $\mathcal{C}$ of cores of an instance $\mathcal{F}$

linear inequality $\sum_{(\neg b) \in \kappa} b \geq 1$. We will mostly treat cores as clauses over (or sets of) blocking variables. Given a set $\mathcal{C}$ of cores, a hitting set $hs \subset \mathcal{B}(\mathcal{F})$ is a set of blocking variables that has non-empty intersection with each $\kappa \in \mathcal{C}$. A hitting set hs is minimum-cost if $\textsc{cost}(\mathcal{F}, hs) \leq \textsc{cost}(\mathcal{F}, hs')$ holds for all hitting sets over $\mathcal{C}$. IHS-based algorithms to MaxSAT rely on the well-known fact that hitting sets over sets of cores provide lower bounds on the optimal cost of instances.

▶ **Proposition 1.** *Let hs be a minimum-cost hitting set over a set $\mathcal{C}$ of cores of an instance $\mathcal{F}$. Then $\textsc{cost}(\mathcal{F}, hs) \leq \textsc{cost}(\mathcal{F})$.*

An important remark to make here for understanding the IHS approach to MaxSAT solving is that Proposition 1 holds for *any* set of cores of an instance. For an minimum-cost hitting set hs over the set $\mathcal{C}$ of *all cores of* $\mathcal{F}$, it holds that $\textsc{cost}(\mathcal{F}, hs) = \textsc{cost}(\mathcal{F})$ and there is a solution τ to $\mathcal{F}$ that sets all blocking variables not in hs to 0.

▶ **Example 2.** Consider the MaxSAT instance $\mathcal{F}$ with $\mathrm{H} = \{(b_1 \vee b_X), (b_2 \vee b_X), (b_3 \vee b_X)\}$ and $\mathcal{B}(\mathcal{F}) = \{b_1, b_2, b_3, b_X\}$ with $w_1(b_1) = w_1(b_2) = w_1(b_3) = 1$ and $w_1(b_X) = 2$. An optimal solution τ_1 to $\mathcal{F}$ is $\tau_1 = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$ and has $\textsc{cost}(\mathcal{F}, \tau) = \textsc{cost}(\mathcal{F}) = 2$. The instance has three subset-minimal cores (MUSes): $\kappa_1 = \{b_1, b_X\}$, $\kappa_2 = \{b_2, b_X\}$ and $\kappa_3 = \{b_3, b_X\}$. For a set $\mathcal{C} = \{\kappa_1, \kappa_2\}$ of cores, an example minimum-cost hitting set hs_1 is $\{b_X\}$ which has $\textsc{cost}(\mathcal{F}, hs_1) = 2 \leq \textsc{cost}(\mathcal{F})$. If we instead have $w_2(b_1) = w_2(b_2) = w_2(b_3) = 1$ and $w_2(b_X) = 4$, then an optimal solution τ_2 is $\tau_2 = \{b_1, b_2, b_3, \neg b_4\}$ and has $\textsc{cost}(\mathcal{F}, \tau_2) = 3$. Now a minimum-cost hitting set hs_2 over $\mathcal{C}$ is $hs_2 = \{b_1, b_2\}$ which has $\textsc{cost}(\mathcal{F}, hs_2) = 2$. Notice that changing weights can significantly alter the minimum-cost hitting sets. Specifically, hs_1 is not minimum-cost w.r.t. w_2 while hs_2 is also a minimum-cost hitting set w.r.t. w_1

3 The Implicit Hitting Set Approach to MaxSAT Solving

Algorithm 1 details IHS, the implicit hitting set algorithm to computing an optimal solution to a single MaxSAT instance $\mathcal{F}$. In short, the algorithm decouples MaxSAT solving into separate core extraction (the Extract-Cores subroutine) and an optimization steps (the Min-Hs subroutine). The core extraction makes use of a SAT solver to extract an increasing number of cores, which are stored in the set $\mathcal{C}$. As a by-product, the procedure also computes a solution τ to $\mathcal{F}$. The solution allows refining the upper bound ub on $\textsc{cost}(\mathcal{F})$, i.e., comparing $\textsc{cost}(\mathcal{F}, \tau)$ to the known upper bound ub and updating it if the new solution has lower cost. The optimization steps compute a minimum-cost hitting set hs over the set $\mathcal{C}$ of cores extracted so far using an IP solver. By Proposition 1 the cost $\textsc{cost}(\mathcal{F}, hs)$ of such a set is a lower bound lb on $\textsc{cost}(\mathcal{F})$. `IHS` terminates once $lb = ub$ and returns τ_{best}, the solution for which $\textsc{cost}(\mathcal{F}, \tau_{best}) = ub$, which is guaranteed to be an optimal solution.

In more detail, when invoked on an instance $\mathcal{F}$, `IHS` begins by initializing the lower bound lb to 0, the upper bound ub to ∞, the best known model τ_{best} to $\varnothing$ and a set $\mathcal{C}$ of cores of $\mathcal{F}$ (represented as sets of blocking variables) to $\emptyset$ (Lines 2-3). Then the main search loop (Line 4-11) is started. During each iteration of the loop, a hitting set hs over $\mathcal{C}$ is computed on Line 5 by solving the integer program detailed in Figure 1 via the procedure `Min-Hs`$(\mathcal{B}(\mathcal{F}), \mathcal{C})$. The procedure returns a minimum-cost set of blocking variables hs that contains at least one variable from each $\kappa \in \mathcal{C}$, i.e., a minimum-cost hitting set over $\mathcal{C}$. The cost $\textsc{cost}(\mathcal{F}, hs)$ of the set is then used to update 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 criteria is checked on Line 7. If the known upper bound matches the known lower bound, the algorithm terminates and returns the current best solution τ_{best}. Otherwise, the core extraction step `Extract-Cores` is invoked on Line 8. The procedure uses a SAT solver iteratively in order to extract previously unseen cores of $\mathcal{F}$ in the form of a disjoint set K of cores s.t. each $\kappa \in K$ is a subset of $\mathcal{B}(\mathcal{F}) \setminus hs$. The cores are extracted using the assumption interface offered by most modern SAT solvers [13, 8] that allows inputting a CNF formula F and a set $\mathcal{A}$ of assumptions in the form of literals. The SAT solver then solves the formula $F \wedge \bigwedge_{l \in \mathcal{A}}(l)$ and returns either (i) a model τ of F that sets $\tau(l) = 1$ for all $l \in \mathcal{A}$ or (ii) a subset $\mathcal{A}_s \subset \mathcal{A}$ such that $F \wedge \bigwedge_{l \in \mathcal{A}_s}(l)$ is unsatisfiable (which is equivalent to F entailing the clause $\{\neg l \mid l \in \mathcal{A}_s\}$).

`Extract-Cores` invokes the internal SAT solver on the hard clauses H under the assumptions $\{\neg b \mid b \in \mathcal{B}(\mathcal{F}) \setminus hs\}$. If the SAT solver reports unsatisfiability, the subset of assumptions returned by the SAT solver corresponds to a core of $\mathcal{F}$. The literals in the core are then removed from the assumptions and the SAT solver reiterated. The procedure terminates when K is a maximal set of disjoint cores over $\mathcal{B}(\mathcal{F}) \setminus hs$ and returns K and τ, the final model returned by the SAT solver that satisfies the hard clauses and all soft clauses that are not in hs nor any of the cores in K.

Since τ satisfies H, it is a solution to $\mathcal{F}$. Thus its cost $\textsc{cost}(\mathcal{F}, \tau)$ is compared to the current upper bound ub and updated if needed on Line 9. If the new 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 here 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. With this intuition. the termination of `IHS` follows by the finite number of cores and hitting sets of $\mathcal{F}$. A detailed argument for the correctness of `IHS` can be found in [3].

▶ **Example 3.** Invoke `IHS` on the instance $\mathcal{F}$ from Example 2 with $w(\mathcal{F}) = w_1$. In the first iteration, the set $\mathcal{C}$ of cores is empty, so `Min-Hs` returns an empty hitting set $hs = \emptyset$ which does not allow increasing the lower bound. At this point $0 = lb < \infty = ub$ so `IHS` does not terminate but instead moves on to `Extract-Cores` to extract a disjoint set of cores over $\mathcal{B}(\mathcal{F}) \setminus hs = \{b_1, b_2, b_3, b_X\}$. There are a number of different possibilities that could be returned. However, all of them contain at most one core that contains at least one of the variables b_1, b_2 or b_3. Say `Extract-Cores` returns $K = \{b_1, b_X\}$ and $\tau = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$.

Since $\textsc{cost}(\mathcal{F}, \tau) = 2 < \infty = ub$ the upper bound is updated to 2 and the best model τ_{best} is set to τ. At this point, `IHS` has found an optimal solution to $\mathcal{F}$. However, since $lb = 0 \neq 2 = ub$ the algorithm does not terminate, but instead augments $\mathcal{C}$ with $\{b_1, b_X\}$ and reiterates. Informally speaking, the optimality of τ_{best} has not been proven yet.

In the next iteration `Min-Hs` is invoked with $\mathcal{C} = \{\{b_1, b_X\}\}$. There exists one minimum-cost hitting set $hs = \{b_1\}$ over $\mathcal{C}$. This hitting set allows refining $lb = 1$ and `Extract-Cores` to extract one more core that is a subset of $\mathcal{B}(\mathcal{F}) \setminus hs = \{b_2, b_3, b_X\}$, say $\{b_2, b_X\}$. In the next iteration, `Min-Hs` computes either $hs = \{b_1, b_2\}$ or $hs = \{b_X\}$. In both cases $lb = \textsc{cost}(\mathcal{F}, hs) = 2 = ub$ so the algorithm terminates, and returns τ_{best}.

We end this section by discussing abstract cores, a recently proposed improvement to `IHS` [8]. In short, an abstract core is a compact representation of a large—potentially exponential—number of regular cores that the `IHS` algorithm can reason with more efficiently. In more detail, an abstraction set $ab \subset \mathcal{B}(\mathcal{F})$ is a subset of n blocking variables 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 clause κ that: (i) contains only blocking variables or count variables and (ii) is entailed by the conjunction of hard clauses of $\mathcal{F}$ and the definitions of count variables. Following [8] we require that all of the blocking variables assigned to the same abstraction set ab have the same weight. This allows the count variables of ab to have well-defined weights; each count variable of ab being assigned to 1 corresponds to one more $b \in ab$ also being assigned to 1, incurring $w(b)$ more cost.

For some intuition on their usefulness, note that an abstract core κ containing a count variable $ab.c[i]$ corresponds to $\binom{|ab|}{|ab|-i+1}$ non-abstract cores where the count variable $ab.c[i]$ variable is exchanged with any subset of ab containing $|ab| - i + 1$ elements. More details can be found in [8].

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 blocking variables as nodes and an edge between any two variables 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 variables 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 variables 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 are monitored. If the extracted (abstract) cores are not driving up the lb computed by the optimizer (`Min-Hs`), then the graph G is reclustered by merging the nodes in the current clusters and then re-clustering the graph. Note that after re-clustering, one single node in G might correspond to several blocking variables of $\mathcal{F}$.

4 MaxSAT with Changing Weights

We move on to our proposal for extending the IHS approach to MaxSAT for computing optimal solutions to MaxSAT instances under changing weights. After formulating in more detail the incremental problem setting we consider, we will describe an extension of `IHS` capable of solving sequences of MaxSAT instances with different weights in an incremental fashion.

```
IHS-INC($\mathcal{F}$, next-w, $k$)
    $\mathcal{C} \leftarrow \emptyset$    $AB \leftarrow \emptyset$;
    $\tau_{best} \leftarrow$ SAT(H($\mathcal{F}$));
    for $i = 1, \ldots, k$ do
        if $i > 1$ then $w(\mathcal{F}) =$ next-w();
        deactivate-abs($AB, w(\mathcal{F})$);
        $ub \leftarrow$ COST($\mathcal{F}, \tau_{best}$);
        $\tau_{best} \leftarrow$ ihs-abscores($\mathcal{F}, \mathcal{C}, AB, ub$);
        output $\tau_{best}$;
```

Algorithm 2 IHS-INC for computing optimal solutions to k different instances of different weights.

4.1 Problem Formulation

Given a MaxSAT instance $\mathcal{F}$ and k different weights w_i for the soft clauses in $\mathcal{F}$, our objective is to compute k solutions $\tau_1 \ldots \tau_k$ to $\mathcal{F}$ such that each τ_i is an optimal solution w.r.t. the weights w_i. We do not put any requirements on how the weights are computed. Our solution algorithm solves the problem sequentially. In particular, the ith weights w_i can depend on the optimal solutions $\tau_1 \ldots \tau_{i-1}$ computed in previous iterations. More formally, we assume that the first weights w_1 are given as part of the input and abstract the computation of all other weights to a black-box oracle `next-w` that is assumed to have access to all information (optimal solutions, previous weights, etc.) from previous iterations.

4.2 Incremental IHS for MaxSAT with Changing Weights

Algorithm 2 details `IHS-INC`, an extension of the IHS algorithm `ihs-abscores` with abstract cores, for solving a MaxSAT instance $\mathcal{F}$ under k different weight functions. The algorithm takes as input the instance $\mathcal{F}$, a function `next-w` for computing the weight functions used in subsequent iterations and k, the number of iterations required. After initializing a set $\mathcal{C}$ of cores and a set AB of abstraction sets on Line 2 as well as obtaining an initial solution τ_{best} by invoking a SAT solver on the hard clauses of $\mathcal{F}$ on Line 3, the algorithm enters its main search loop (Lines 4-9).

In each iteration of the loop, the algorithm computes an optimal model w.r.t. the ith weights. Each iteration starts with the new weights being obtained on Line 5 and an initial upper bound ub computed from the current best model τ_{best} on Line 7. In the first iteration, τ_{best} will be the model obtained by checking the satisfiability of the hard clauses (on Line 3). In subsequent iterations, τ_{best} will be the optimal model computed in the previous iteration. Afterwards, an optimal model w.r.t. the current weights is computed using the function `ihs-abscores` implementing the IHS algorithm with abstract cores for computing one optimal solution to the instance.

A central fact to note in `IHS-INC` is that—on every iteration except the first one—`ihs-abscores` is invoked with a set $\mathcal{C}$ of cores and AB of abstraction sets that are *non-empty*. Indeed, all of the cores and abstract cores that are computed during previous iterations are preserved and used in subsequent iterations as well. Similarly, as many abstraction sets as possible are also preserved between iterations. Recall that `ihs-abscores` assumes that all blocking variables assigned to the same abstraction set ab have the same weight. As the weights of blocking variables can change between iterations, we stop extracting new abstract cores over ab if the weights of the blocking variables in ab are changed to be unequal. More precisely, we say that an abstraction set ab is valid if $w(b_i) = w(b_j)$ holds for any $b_i, b_j \in ab$.

In Algorithm 2 the `deactivate-abs` method loops over the set AB to check which ones are not valid anymore. The `ihs-abscores` method then only extracts abstract cores over valid abstraction sets. However, since the definition of an abstract core is independent from the weights of blocking variables, abstract cores containing count variables in an invalid ab are still preserved and used in subsequent iterations, the definitions of count variables of invalid abstraction sets are kept in the SAT and IP solvers. We also allow blocking variables from invalid abstraction sets to be assigned to other abstraction sets in later iterations. More specifically, the blocking variables from invalid abstraction sets are reintroduced into the graph and allowed to be clustered in later iterations.

The correctness of `IHS-INC` follows from the fact that the definition of a core and an abstract core depends only on the clauses in $\mathcal{F}$, and the clauses defining count variables. Neither of these change between iterations so all of the cores computed in previous iterations can be kept in subsequent ones.

▶ **Example 4.** Invoke `IHS-INC` on the instance $\mathcal{F}$ from Example 2 and assume the weight function w_1 for the first instance in a sequence of instance to be solved. To keep the example simple, we also assume that no abstraction sets or abstract cores are used in the execution.

Assume that the initial SAT solver call on Line 3 on the clauses of $\mathcal{F}$ obtains a model $\tau_{best} = \{b_1, b_2, b_3, \neg b_X\}$ and an initial upper bound $ub = \text{COST}(\mathcal{F}, \tau_{best}) = 3$. The algorithm then invokes `ihs-abscores` with $\mathcal{C} = \emptyset$ and $ub = 3$. As `ihs-abscores` without abstract cores corresponds exactly to `IHS` detailed in Algorithm 1, Example 3 details one possible execution when solving $\mathcal{F}$. After that execution, the procedure returns $\tau_{best} = \{\neg b_1, \neg b_2, \neg b_3, b_X\}$ and updates $\mathcal{C} = \{\{b_1, b_X\}, \{b_2, b_X\}\}$.

Assume then that the weights of $\mathcal{F}$ are updated to w_2 as detailed in Example 2. The new weights are then used to update the upper bound to $\text{COST}(\mathcal{F}, \tau_{best}) = 4$ before `ihs-abscores` is invoked again. In the first iteration of the search loop of `ihs-abscores`, the set $\mathcal{C}$ already contains two cores. As such `Min-Hs` returns the minimum-cost hitting set $hs = \{b_1, b_2\}$ and updates $lb = \text{COST}(\mathcal{F}, hs) = 2$. Afterwards, `Extract-Cores` extracts the core $\{b_3, b_X\}$ and returns (for example) the solution $\tau = \{b_1, b_2, b_3, \neg b_X\}$. This solution has $\text{COST}(\mathcal{F}, \tau) = 3$, so the ub and τ_{best} is updated. In the next iteration, `Min-Hs` computes the hitting set $hs = \{b_1, b_2, b_3\}$ which has $\text{COST}(\mathcal{F}, hs) = 3$ and allows the algorithm to terminate.

Example 4 demonstrates how `IHS-INC` is able to solve the second iteration just by extracting one more core. In contrast, it can be shown that restarting the search from scratch (i.e., invoking `IHS`) results in at least 3 cores being extracted when solving $\mathcal{F}$ with $w(\mathcal{F}) = w_2$ from Example 2.

4.3 Realizing IHS-INC

On an abstract level, as demonstrated by Algorithm 2, `IHS-INC` is relatively straightforward to implement given a procedure for `ihs-abscores`. However, in reality, the engineering aspects are less trivial. We continue by detailing our implementation which is built on top of MaxHS [11, 8], a state-of-the-art IHS MaxSAT solver. In practice this requires several changes to the underlying data structures and procedures of MaxHS, especially those concerning the internal representation of soft clauses and their weights.

4.3.1 Handling Weight Changes

Our goal is to provide an API function `changeWeight(`i, w`)` which can be called incrementally to change the weight of the ith input soft clause to $w \geq 0$. The necessary changes to MaxHS are applied to the following components.

WCNF Simplification

Before solving, MaxHS performs a series of simplifications to the input instance. In particular, after simplifying, the list of soft clauses is not in general equal to the soft clauses of the input instance[1]; since some soft clauses are removed due to either always being satisfied or impossible to satisfy given the hard clauses, and some soft clauses are merged. In order to have access to `changeWeight(`i, w`)`, we implement a mapping currentIndex which takes an index of the original soft clauses as input, and returns either: (a) an index of the internal list of the soft clauses (note that since some soft clauses have been merged, the same index may correspond to several indices of the input instance), (b) `SAT` if the soft clause has been removed since it is implied by the hard clauses or (c) `UNSAT` if the soft clause has been removed because it is unsatisfiable given the hard clauses. Furthermore, we also keep track of all preimages of this map in order to perform updates to it correctly. After simplifying, currentIndex will remain constant. During simplification, MaxHS also computes baseCost as the sum of the weights of soft clauses which cannot be satisfied, and totalWeight as the sum of the weights of soft clauses remaining after simplification. These numbers may also naturally change due to changing weights of the original instance.

In more detail, the simplification procedures are the following:

- *Hardening of soft clauses.* MaxHS checks the input weights of the soft clauses and determines whether some soft clauses can be hardened due to their high weight. Since in our setting such a high weight may change to an arbitrarily low one, this feature is disabled.
- *Unit hard clauses and equalities.* MaxHS performs unit propagation over the hard clauses, checks for equalities implied by the hard clauses, and performs pure literal elimination. Although these procedures do not concern the weights, they may modify the original list of soft clauses. In particular, a soft clause may be satisfied due to e.g. containing a literal which has been assigned to true via unit propagation, in which case the soft clause is removed; we update currentIndex by setting the corresponding index to `SAT`. If a soft clause becomes empty due to e.g. containing only literals which have been assigned to false via unit propagation, it is removed and baseCost is updated; we update currentIndex by setting the corresponding index to `UNSAT`. Finally, tautologies are removed; for a (tautological) soft clause we set the corresponding index to `SAT`.
- *Contradictory unit clauses.* If there is a pair of contradictory unit clauses one of which is soft and the other is hard, the soft clause is falsified, so we update currentIndex by setting its index to `UNSAT`. If there is a pair of contradictory soft unit clauses, the base-version of MaxHS would only preserve the clause with higher weight, setting its new weight as the difference and incrementing baseCost with the smaller weight. In our setting we need to preserve both; we additionally set the new weight of the lower-weight clause to zero, and keep track of such contradictory unit soft clauses within the contradictoryUnit datastructure. In particular, MaxHS initializes blocking variables in such a way that unit soft clauses are used as blocking variables, and new variables are declared only for non-unit softs. In contrast, we declare new blocking variables for unit soft clauses for which contradictoryUnit is true.
- *Duplicate clauses.* If there is a pair of duplicate clauses with a hard clause and a soft clause, the soft clause is subsumed by the hard clause. In this case, we update currentIndex at

[1] Note that, in contrast to the pseudocode, MaxHS does not assume that every soft clause is a unit negation of a blocking variable. Instead the solver maintains the full clause and declares a blocking variable for it internally.

```
changeWeight($(i, w)$)
    $\delta \leftarrow w -$ originalWeights[$i$];
    if currentIndex[$i$] $=$ $UNSAT$ then
        baseCost $\leftarrow$ baseCost $+ \delta$;
    else if currentIndex[$i$] $=$ $SAT$ then
        return;
    else
        if contradictoryUnit[currentIndex[$i$]] then
            resolve unit softs;
        else
            totalWeight $\leftarrow$ totalWeight $+ \delta$;
            weights[currentIndex[$i$]] $\leftarrow$ weights[currentIndex[$i$]] $+ \delta$;
    originalWeights[$i$] $\leftarrow w$;
    update CPLEX;
```

Algorithm 3 Procedure for changing the weight of the ith soft clause to w.

the corresponding index to `SAT`. If there is a duplicate of two soft clauses, they are joined into one by setting the weight as the sum of the two weights. We update currentIndex by setting the index of the removed soft clause to the same index as the preserved one.

- *Flipping literals.* If there is a unit soft clause with a positive literal, that literal is flipped in the instance in order to use it as a blocking variable (so that setting the blocking variable to true incurs the cost of the soft clause). We only do this in the case that the soft clause does not have a contradictory unit soft clause.

Note that hardening of soft clauses is disabled since it may change the set of cores of the instance being solved and lead to `IHS-INC` computing cores that are not valid to preserve between iterations.

▶ **Example 5.** Consider the instance $\mathcal{F}$ with $\mathrm{H} = \{(b_1 \lor b_2)\}$ and $\mathcal{B}(\mathcal{F}) = \{b_1, b_2\}$ with $w(b_1) = 1$ and $w(b_2) = 10$. During hardening, MaxHS invokes a SAT solver on H in hopes of finding a good model that allows hardening of soft clauses. Assume that the model $\tau = \{b_1, \neg b_2\}$ is computed. Since $\textsc{cost}(\mathcal{F}, \tau) = 1 < w(b_2)$ MaxHS concludes that the soft clause $(\neg b_2)$ can be hardened and invokes `ihs-abscores` on the instance $\mathcal{F}^H = \{(b_1 \lor b_2), (\neg b_2)\}$ with $\mathcal{B}(\mathcal{F}^H) = \{b_1\}$. While the optimal solutions of both $\mathcal{F}$ and $\mathcal{F}^H$ are the same, the set of cores are not, $\kappa^H = \{b_1\}$ is an example of a core of $\mathcal{F}^H$ that is not a core of $\mathcal{F}$. In other words, κ^H could not in general be preserved between the iterations of `IHS-INC` as it is not a core of any instance where $(\neg b_2)$ can not be hardened.

CPLEX Interface

The underlying IP solver CPLEX, used for solving the hitting set problems, has to be updated between iterations with the new sequence of weights. We implement this within the CPLEX interface of MaxHS by using `CPXXchgcoef`[2] to change the coefficient of the objective function corresponding to the weight of a blocking variable. This update is performed only if the corresponding blocking variable exists in CPLEX.

[2] https://www.ibm.com/docs/en/icos/12.10.0?topic=c-cpxxchgcoef-cpxchgcoef

The resulting procedure for `changeWeight(`i, w`)` is detailed as Algorithm 3. We compute δ as the difference between the new weight w and the current weight stored in originalWeights[i] (line 2). If currentIndex[i] is `UNSAT`, it suffices to increment baseCost by δ (lines 3 and 4). If currentIndex[i] is `SAT`, we simply do nothing (lines 5 and 6). Otherwise currentIndex[i] contains the index of the corresponding internal soft clause. Now, if this internal soft clause has a contradictory unit clause, we resolve these two unit soft clauses (lines 8 and 9). Otherwise, we increment totalWeight and the internal weight weights[currentIndex[i]] by δ (lines 11 and 12). Finally, we set originalWeights[i] to the new weight w (line 13) and perform the necessary updates to CPLEX (line 14).

4.3.2 Weight-based Reasoning

In addition to correctly taking into account the simplification procedure and updating the IP solver, during solving MaxHS performs weight-based reasoning, which either has to be disabled or reimplemented by taking into account that weights may potentially change. These reasoning procedures are the following.

Reduced Cost Fixing

MaxHS considers the linear programming (LP) relaxation of the hitting set problem, and using so-called reduced costs corresponding to the optimal solution of the LP, determines whether a soft clause can be hardened [3]. In particular, this is determined via the optimal cost of the LP, the reduced cost corresponding to the blocking variable, and the cost of a feasible solution to the MaxSAT problem. After changing weights, all of these numbers may change arbitrarily. Hence, it is clear that soft clauses hardened due to reduced cost fixing may invalidate the current instance and alter the set of cores the instance (recall the earlier discussion on hardening). Due to this, reduced cost fixing is disabled.

Abstract Cores: Graph and Totalizers

Recall that in order to determine which blocking variables occur in cores often together, MaxHS constructs a weighted undirected graph based on the accumulated cores [8]. Nodes of this graph correspond to partitions of the set of blocking variables, and weights of the edges between nodes to how many times the blocking variables occur together in a core. In particular, it is assumed that blocking variables within a node and in adjacent nodes have the same weight. In order to preserve these invariants, if a node contains several blocking variables which now have different weights, the node is removed from the graph. Similarly, if the incident nodes of an edge contain blocking variables with different weights, the edge is removed from the graph.

In order to encode cardinality constraints over count variables corresponding to abstract cores, MaxHS makes use of totalizers [7]. It is assumed that blocking variables used as inputs of a totalizer have the same weight. Furthermore, each totalizer is assigned this weight in order to compute new lower bounds. In order to preserve this invariant, we check which totalizers contain inputs whose weights have changed. If all weights have changed to the same new weight, we simply reset the current weight of the totalizer to the new weight. If weights are different, the totalizer is invalid, so it is removed, and so are all totalizers containing a subset of the inputs of the totalizer.

4.3.3 Solving Procedure

With all of this in place, for solving the instance at iteration $i > 1$ we recompute the sum of the weights of soft clauses known to be satisfiable from the existing model in the SAT solver (τ_{best} in Algorithm 2)—this weight is used to determine the upper bound by subtracting it from totalWeight. Furthermore, we recompute the sum of the weights of blocking variables that are fixed to true by the SAT solver, and set the lower bound to this number. After reinitializing the graph and totalizers related to abstract cores, we may solve the updated instance as usual.

In addition to the techniques discussed in this section, our implementation of `IHS-INC` also makes use of a number of previously proposed heuristics for extracting a large number (hundreds) of cores from each hitting set [12, 10, 28]. The techniques that have not been discussed in this section are all sound to keep between iterations. Recall that, as long as a core extraction heuristic computes a set of blocking variables is a core of the current instance, the same set will be a core in subsequent iterations as well.

5 Experimental Evaluation

In this section we provide an empirical evaluation comparing MaxHS (MaxSAT evaluation 2020 version[3]) to our implementation of `IHS-INC` built on top of MaxHS. The non-incremental MaxHS is run with default parameters, with all its optimizations including hardening of soft clauses during simplification and in the form of reduced cost fixing enabled.

All experiments were run on 2.60-GHz Intel Xeon E5-2670 8-core machines with 64GB memory and CentOS 7. We set a per-instance time limit of 7200 seconds (2 hours) and a memory limit of 16 GB. Specifically, the 7200-second time limit is for solving a single instance n times with n different weights $w_1, \ldots, w_n$. 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) is considered as a timeout if $\sum_{i=1}^{k} t_i > 7200$ seconds.

We consider two different recently-proposed methods for learning interpretable classifiers, namely decision trees and decision rules, via MaxSAT. The input for both scenarios is a dataset $(\mathbf{X}_i, y_i)$, $i = 1, \ldots, n$, of binary examples $\mathbf{X}_i \in \{0,1\}^m$ and classes $y_i \in \{0,1\}$. Each coordinate $j = 1, \ldots, m$ of an example $\mathbf{X}_i = (x_i^1, \ldots, x_i^m)$ is called a feature. The goal is to learn a binary classifier (a function mapping each example in the feature space $\{0,1\}^m$ to a class in $\{0,1\}$) which minimizes the training error consisting of the number of misclassified examples in the input dataset. In the context of changing weights, we consider two different methods designed to avoid overfitting. For decision trees, AdaBoost [16] is an algorithm where a sequence of shallow decision trees are learned by iteratively changing the weights of the examples in the training set. For decision rules, we include a regularizing term to the objective function which also enforces the sparsity of the resulting rule [18], and iteratively vary the value of the regularization parameter.

5.1 Case Study 1: MaxSAT for Boosting Decision Trees

Decision trees are classifiers with the structure of a full binary tree for binarized data. Leaf nodes are associated with a particular class (in our setting, 0 or 1), and non-leaf nodes with a feature $j = 1, \ldots, m$. An example $\mathbf{X} = (x_1, \ldots, x_m)$ is classified by starting from the root node, checking the value of x_j for the feature j associated to the node, and proceeding

[3] https://maxsat-evaluations.github.io/2020/descriptions.html

Table 1 Statistics on MaxSAT instances used for AdaBoost.

	Minimum	Maximum	Average	Median
Number of variables	2129	26396	8310.4	7768
Number of hard clauses	8176	96382	30343.8	26772
Literals in hard clauses	38524	5342964	718654.9	211629
Average length of hard clauses	3.45476	70.1593	16.4	10.3187
Number of soft clauses	27	6473	803.6	342

recursively to the left child if $x_j = 0$ and to the right child if $x_j = 1$. The class is then determined by the leaf node which terminates the recursion.

We consider the MaxSAT encoding for learning a decision tree of depth at most U [16], based on a SAT encoding for learning a decision tree with exactly N nodes [26]. In addition to variables and hard clauses for encoding the structure of a valid binary tree, its depth, and the classification of the training data, the MaxSAT encoding has variables b_i for each example $\mathbf{X}_i$ with the interpretation that b_i is true if and only if example $\mathbf{X}_i$ is classified correctly. The objective is then to minimize the training error via unit-weight soft clauses (b_i) for each example $\mathbf{X}_i$. An instance formed from a dataset with n examples and m features has $O(n+m)$ variables, $O(n+m)$ hard clauses of length $O(m)$, and n unit soft clauses.

In particular, here we focus on the implementation of AdaBoost [14], an ensemble method where multiple weak classifiers (in this context, shallow decision trees) are trained and then combined into a single classifier via a weighted voting scheme. This is achieved via changing the weights of the soft clauses iteratively [16]. In more detail, after learning a decision tree, the weight $w(b_i)$ for each $i = 1, \ldots, n$ is updated via

$$\widehat{w}(b_i) = \frac{w(b_i) \cdot f_i}{\sum_{j=1}^{n} w(b_j) \cdot f_j}$$

where $f_i = \exp(-\alpha)$ if the ith example was classified correctly and $\exp(\alpha)$ otherwise, $\alpha = \frac{1}{2}\ln(\frac{1-\varepsilon}{\varepsilon})$, and ε is the training error. As long as $\varepsilon < 0.5$, this raises the weight of incorrectly classified examples and lowers the weight of correctly classified examples. Intuitively, the following decision tree will consider that misclassified examples are more important than correctly classified ones. Then, weights are discretized by setting

$$w(b_i) = \text{round}\left(\frac{\widehat{w}(b_i)}{\min_{j=1,\ldots,n} \widehat{w}(b_j)}\right).$$

In other words, if an example is classified correctly at each iteration, its corresponding weight will remain constant 1 due to discretization. If an example is classified incorrectly at each iteration, its weight will grow exponentially.

In contrast to using an incomplete MaxSAT solver by starting it from scratch at each iteration [16], we consider solving each iteration exactly in an incremental fashion. For benchmarks, we take the 15 datasets used in [16] (which were downloaded from CP4IM[4] and discretized[5]). For the exact number of examples and the number of features in these instances, we refer the reader to [16, Table 1]. For each dataset, we generated different training sets by taking 20%, 30%, ..., 80% of the available data, resulting in 105 training

[4] `https://dtai.cs.kuleuven.be/CP4IM/datasets/`
[5] `https://gepgitlab.laas.fr/hhu/maxsat-decision-trees`

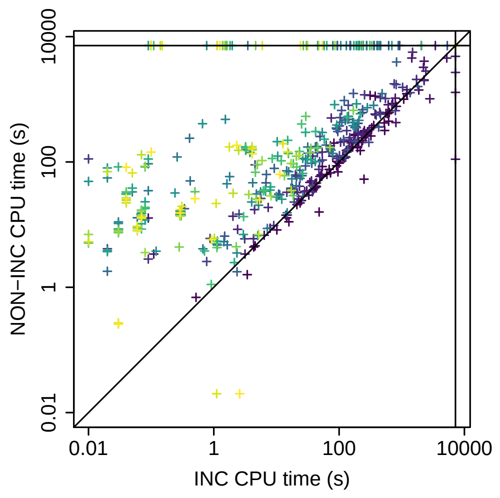

Figure 2 Incremental vs. non-incremental MaxHS for AdaBoost. Each point is an iteration.

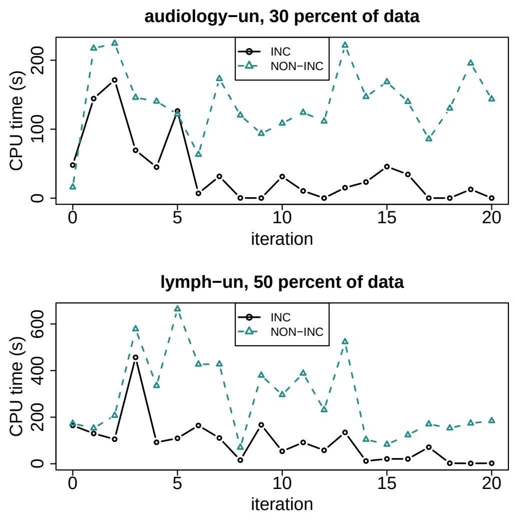

Figure 3 Incremental vs. non-incremental MaxHS for AdaBoost on example datasets.

sets. We set the maximum depth to $U = 2$ and used 21 iterations for AdaBoost. Detailed statistics on the resulting MaxSAT instances are provided in Table 1.

The results are summarized in Figure 2, where each point is a single iteration, the x-axis is the CPU time-consumed by our implementation, and the y-axis is the CPU time-consumed by MaxHS. Points are colored by the iteration number: the higher the iteration number, the more yellow the point (and the lower, the more blue). We clearly see that almost all lower iterations take approximately the equal amount of time, with a few more timeouts exhibited by our implementation than by MaxHS (four points on the right border of the plot). However, for higher iterations, we see a clear improvement from using the incremental version of MaxHS. In particular, some iterations which take 100 seconds to solve using MaxHS are now solved in a matter of seconds, and MaxHS exhibits a significant number of timeouts which are solved using the incremental version (points on the upper border of the plot).[6]

Dataset specific examples of how the runtimes of non-incremental and incremental MaxHS differ when iterating over the sequences of instances the datasets give rise to are provided in Figure 3. We observe that for almost all iterations, the performance of the incremental version is significantly better than that of the standard non-incremental MaxHS. This is the case in particular for the later iteration; evidently, incremental computations start paying off noticeably after solving the first few instances in the sequence.

5.2 Case Study 2: MaxSAT for Learning Decision Rules

Decision rules are classifiers which take the simple and interpretable form of if-then-else rules. Here we consider MLIC [18], a framework for learning decision rules via MaxSAT, in particular decision rules where the implicant is a CNF formula $\mathcal{R}$ over the features containing exactly K clauses, and the consenquent is simply "class is 1". In addition to minimizing

[6] We also tried using the previous optimal solution to calculate an initial UB in non-incremental MaxHS, but did not observe any significant performance improvements. Note that MaxHS only terminates when the upper bound equals the lower bound, even given an optimal solution, MaxHS has to prove its optimality by extracting cores which yield a hitting set of the same cost.

Table 2 Data on MaxSAT instances used for MLIC.

	Minimum	Maximum	Average	Median
Number of variables	182	183105	26122.9	2339.5
Number of hard clauses	523	49086727	3792656.9	47721
Literals in hard clauses	1473	120037762	9765213.2	216129
Average length of hard clauses	2.21576	132	15.8	2.5
Number of soft clauses	157	48139	8224.0	1627.5

the training error, the goal is to learn *sparse* decision rules. Sparsity of the learned rules is enforced by a regularizing term. In particular, the objective is to minimize $\lambda|\mathcal{E}_{\mathcal{R}}| + \|\mathcal{R}\|$, where $|\mathcal{E}_{\mathcal{R}}|$ is the number of misclassified examples and $\|\mathcal{R}\|$ is the number of literals in $\mathcal{R}$. The choice of the regularization parameter[7] $\lambda > 0$ is a difficult task. A simple method for choosing λ is to perform an exhaustive grid search over an interval and choosing the λ that minimizes e.g. the cross-validated error. Note the form of the objective function, namely minimizing the linear combination of an error and a regularizing term, is very general, and interestingly similar MaxSAT-based methods for learning sparse decision sets [32] and lists [31] also share a similar objective.

The MaxSAT encoding has variables b_k^j for each clause index $k = 1, \ldots, K$ and each feature $j = 1, \ldots, m$, and variables η_i for each example $\mathbf{X}_i$. Here b_k^j is assigned to true if and only if feature j occurs in the kth clause, and η_i is assigned to true if and only if example $\mathbf{X}_i$ is classified incorrectly. In addition to hard clauses encoding the semantics, soft clauses $(\neg\eta_i)$ with unit weights and $(\neg b_k^j)$ with weight λ are used to encode the objective function. An instance resulting from encoding a dataset with n examples and m features has $O(n+m)$ variables, $O(n+m)$ hard clauses with length $O(m)$, and $O(n+m)$ unit soft clauses.

Following [18], we consider computing the optimal decision rules for $\lambda = 0.25, 0.5, \ldots, 5.0$, in this exact order. As we start from a low value of λ, the iterative procedure first learns decision rules that are more sparse and less attention is given to correct classification, and as λ is incremented, more importance is given to classifying the examples correctly than to sparsity. We use the same 10 datasets (from the UCI repository[8]) which were discretized via adapting the script provided by the MLIC repository[9]. For the exact number of examples and the number of features, we refer the reader to [18, Table 1]. For each dataset, we generated training sets by taking 10%, 20%, ..., 90% of the available data, resulting in 90 different training sets. We learned CNF rules consisting of $K = 2, 3$ clauses (as instances with $K = 1$ clauses were observed to be solved directly using the IP solver due to all constraints being seeded into CPLEX). This gave rise to 180 different runs each with 20 iterations. Detailed statistics on the MaxSAT instances are provided in Table 2.

Our results are summarized in Figure 4, where each point corresponds to a single iteration, the x-axis is the CPU time of the incremental version, the y-axis is the CPU time of basic MaxHS, and point are colored by the iteration number. We observe a very clear improvement in favor of the incremental version, especially for higher iterations. MaxHS also exhibits a significant number of timeouts for iterations that are solved using the incremental version.

Dataset specific examples of how the runtimes of non-incremental and incremental MaxHS

[7] Note that, unlike is typically the case, the role of λ in [18] is the weight of the error term, not the sparsity term (which is in fact the regularizer). The sparsity term has coefficient $1/\lambda$.
[8] `https://archive.ics.uci.edu/ml/`
[9] `https://github.com/meelgroup/MLIC/tree/MLIC`

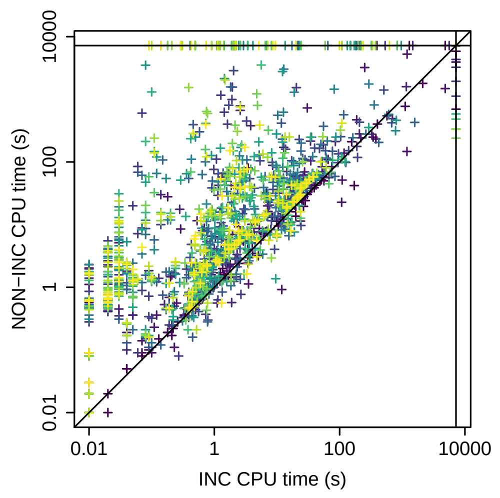

Figure 4 Incremental vs. non-incremental MaxHS for MLIC. Each point is an iteration.

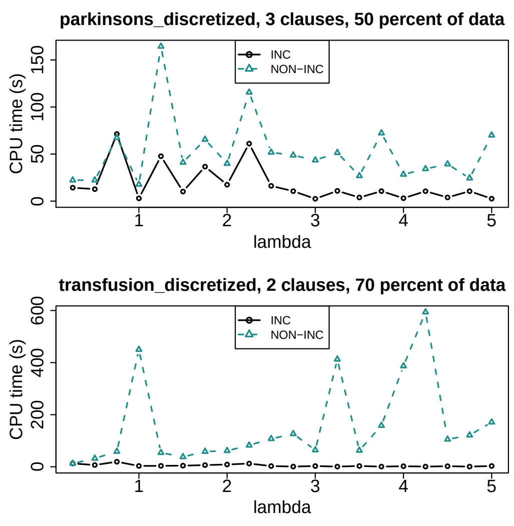

Figure 5 Incremental vs. non-incremental MaxHS for MLIC on example datasets.

differ when iterating over the sequences of instances the datasets give rise to are provided in Figure 5, with the value of lambda (corresponding to the iteration) on the x-axis and the CPU time on the y-axis. While we observe some variation in the runtime for both solvers, e.g. the iterations corresponding to $\lambda = 1.25, 2.25$, are slower to solve, the incremental version is clearly faster on most iterations. Some instances are significantly easier, essentially trivial, to solve using the incremental version compared to the non-incremental solver; the bottom plot in Figure 5 provides such an example.

6 Conclusions

Various types of real-world optimization problems, requiring solving a sequence of related problem instances, call for solvers that can make use of incremental computations across the instances. Motivated by recent applications of MaxSAT solvers, we adapted one of the key MaxSAT solving approaches—the implicit hitting set approach—to cope with incrementality under changes to the weights of soft clauses. While it is seemingly simple to adapt a rudimentary version of the IHS approach to deal with this type of incrementality, the various search techniques applied in MaxHS, a state-of-the-art IHS MaxSAT solver, make such adaptations non-trivial. In particular, we explained which search techniques can and cannot be adapted for incremental computations under changing weights. Taking these observations into practice, we adapted MaxHS to support incrementality under changing soft clause weights. Using two recent real-world applications of MaxSAT in the context of interpretable machine learning as examples, we showed that the incremental version of MaxHS provides significant runtime improvements over MaxHS (despite all of the performance-improving optimizations used in the non-incremental version) when solving sequences of MaxSAT instances with adaptively changing weights. As future work, we aim to generalize the framework further to allow e.g. efficiently altering the set of hard and soft clauses between iterations without increasing the sizes of extracted cores or hitting set instances met.

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. MaxSAT Evaluation 2018: New developments and detailed results. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):99–131, 2019. `doi:10.3233/SAT190119`.

5 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2019: Solver and Benchmark Descriptions*, volume B-2019-2 of *Department of Computer Science Report Series B*. Department of Computer Science, University of Helsinki, Finland, 2019. URL: `https://hdl.handle.net/10138/308068`.

6 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. *Maximum Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 24, pages 929–991. IOS Press, 2021. `doi:10.3233/FAIA201008`.

7 Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003 - 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. `doi:10.1007/978-3-540-45193-8_8`.

8 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`.

9 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`.

10 Jessica Davies. *Solving MAXSAT by Decoupling Optimization and Satisfaction.* PhD thesis, University of Toronto, 2013. URL: `https://hdl.handle.net/1807/43539`.

11 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`.

12 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.

13 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.

14 Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. doi:10.1006/jcss.1997.1504.

15 Bishwamittra Ghosh and Kuldeep S. Meel. IMLI: an incremental framework for MaxSAT-based learning of interpretable classification rules. In Vincent Conitzer, Gillian K. Hadfield, and Shannon Vallor, editors, *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society, AIES 2019, Honolulu, HI, USA, January 27-28, 2019*, pages 203–210. ACM, 2019. doi:10.1145/3306618.3314283.

16 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.

17 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.

18 Dmitry Malioutov and Kuldeep S. Meel. MLIC: A MaxSAT-based framework for learning interpretable classification rules. 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 312–327. Springer, 2018. doi:10.1007/978-3-319-98334-9_21.

19 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.

20 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.

21 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.

22 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.

23 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. 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 564–573. Springer, 2014. doi:10.1007/978-3-319-10428-7_41.

24 António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013. doi:10.1007/s10601-013-9146-2.

25 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`.

26 Nina Narodytska, Alexey Ignatiev, Filipe Pereira, and João Marques-Silva. Learning optimal decision trees with SAT. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1362–1368. ijcai.org, 2018. `doi:10.24963/ijcai.2018/189`.

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

28 Paul Saikko. Re-implementing and extending a hybrid SAT-IP approach to maximum satisfiability. Master's thesis, University of Helsinki, 2015. URL: `http://hdl.handle.net/10138/159186`.

29 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`.

30 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`.

31 Jinqiang Yu, Alexey Ignatiev, Pierre Le Bodic, and Peter J. Stuckey. Optimal decision lists using SAT. *CoRR*, abs/2010.09919, 2020. URL: `https://arxiv.org/abs/2010.09919`, `arXiv:2010.09919`.

32 Jinqiang Yu, Alexey Ignatiev, Peter J. Stuckey, and Pierre Le Bodic. Computing optimal decision sets with SAT. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 952–970. Springer, 2020. `doi:10.1007/978-3-030-58475-7_55`.

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`.