Revisiting SAT-Based Solvers: MaxSAT Rules and Core Sequences

JOSEP ALÒS, Logic & Optimization Group (LOG), University of Lleida, Spain
CARLOS ANSÓTEGUI, Logic & Optimization Group (LOG), University of Lleida, Spain
EDUARD TORRES, Logic & Optimization Group (LOG), University of Lleida, Spain

In this paper, we revisit the state-of-the-art of MaxSAT solving. We focus on SAT-based MaxSAT solving algorithms, mainly on Core-guided MaxSAT solvers. We show how to describe Core-guided solvers with Non-CNF MaxSAT rules plus the *Extension* rule. Equipped with these rules, we show how to apply them alternatively to obtain new Core-guided MaxSAT solvers. Since Core-guided solvers essentially solve a sequence of SAT instances, we also discuss how Core-guided MaxSAT solvers traverse the search space of possible sequences of SAT instances, the existence of exponentially harder sequences, and how to avoid them. The experimental investigation shows comparable and complementary performance to state-of-the-art solvers.

JAIR Associate Editor: Kuldeep Meel

JAIR Reference Format:
Josep Alòs, Carlos Ansótegui, and Eduard Torres. 2026. Revisiting SAT-Based Solvers: MaxSAT Rules and Core Sequences. *Journal of Artificial Intelligence Research* 85, Article 16 (February 2026), 29 pages. DOI: 10.1613/jair.1.19525

1 Introduction

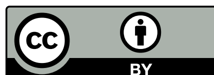
The Maximum Satisfiability (MaxSAT) problem is the optimization variant of the SAT (Satisfiability) problem. In this variant, the goal is to satisfy (or falsify) the maximum (or minimum) number of constraints (clauses) in the given problem. In more specific variants, we have hard constraints (that we must satisfy) and soft constraints (which we can falsify). Several variants of this problem, including weights for soft clauses, are described in the preliminaries section.

There are also different solving techniques or algorithmic approaches depending on whether we can guarantee the optimality of the solution (using exact or complete algorithms) or not (using anytime or incomplete algorithms). While exact algorithms aim to produce an optimal solution, anytime algorithms seek to provide the best possible suboptimal solution as quickly as possible without guaranteeing optimality.

Among the exact algorithms, three main families exist: SAT-based MaxSAT algorithms (which iteratively call a SAT solver) (Ansótegui, Bonet, et al. 2013; Morgado et al. 2013), Branch-and-Bound algorithms (B&B version of a SAT solver) (C. M. Li, Xu, et al. 2021; C.-M. Li et al. 2022), and hybrid algorithms using Mixed Integer Programming (MIP) and SAT solving techniques (Davies and Bacchus 2011). This paper focuses on the first family, which can be further divided into two categories: Core-guided algorithms and Model-guided algorithms.

As our first contribution, we revisit Core-guided algorithms in the context of MaxSAT resolution rules. We describe Core-guided algorithms as complete inference algorithms in the sense that they iteratively apply MaxSAT rules to replace certain constraints (or clauses, in the context of MaxSAT solvers) with another set of constraints. If a rule is complete, then a sequence of applications of that rule will solve the problem.

Authors' Contact Information: Josep Alòs, ORCID: https://orcid.org/0000-0002-7342-2701, josep.alos@udl.cat, Logic & Optimization Group (LOG), University of Lleida, Lleida, Spain; Carlos Ansótegui, ORCID: https://orcid.org/0000-0001-7727-2766, carlos.ansotegui@udl.cat, Logic & Optimization Group (LOG), University of Lleida, Lleida, Spain; Eduard Torres, ORCID: https://orcid.org/0000-0002-3136-7513, eduard.torres@udl.cat, Logic & Optimization Group (LOG), University of Lleida, Lleida, Spain.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2026 Copyright held by the owner/author(s).
DOI: 10.1613/jair.1.19525

More specifically, a Core-guided algorithm queries a SAT solver to check whether all soft and hard constraints can be satisfied. If the SAT solver returns "yes", the optimal solution has been found. Otherwise, the soft constraints within the unsatisfiable core returned by the SAT solver are replaced by a new set of soft constraints using a MaxSAT rule. This new set includes at least one trivially false constraint, allowing the Core-guided algorithm to easily increase the lower bound on the minimum number of soft constraints that can be falsified.

It has not been clear within the community whether Core-guided algorithms truly use MaxSAT rules in practice. In SAT, the Resolution rule (Davis and Putnam 1960) is complete and forms the foundation of modern SAT solvers. In MaxSAT, a corresponding MaxSAT Resolution rule also exists and is complete (Larrosa and Heras 2005). However, this rule is not used in Core-guided MaxSAT solvers. As stated in (Filmus et al. 2020) (where MaxRes refers to the MaxSAT Resolution rule (Larrosa and Heras 2005), and [20] corresponds to citation (Narodytska and Bacchus 2014) in this paper):

"However, since MaxRes needs to maintain a stronger invariant than merely satisfiability, it seems reasonable that for certifying unsatisfiability, MaxRes is weaker than Resolution. (This would explain why, in practice, MaxSAT solvers do not seem to use MaxRes – possibly with the exception of [20], but instead directly call SAT solvers, which use standard resolution)."

There have been some efforts to formalize and explain MaxSAT algorithms from a unified perspective (Fazekas et al. 2018; Ihalainen, Berg, et al. 2024). In this paper [1], we revisit and present MaxSAT rules to explore their connection with Core-guided algorithms. For example, we show that the OLL algorithm (Andres et al. 2012) can also be described using MaxSAT rules.

The MaxSAT rules we introduce may yield conclusions that are not in Conjunctive Normal Form (CNF). Translating these into CNF may require extending the formula with new variables by applying the *Extension* rule. Depending on the specific MaxSAT rule used, how it is applied to the soft constraints in a core, and how the *Extension* rule is employed, we obtain different instantiations — each impacting the efficiency of the Core-guided solver. We report on a particular combination that compares favorably with state-of-the-art Core-guided solvers.

By describing Core-guided MaxSAT algorithms in terms of MaxSAT rules, Extension rules, and the Resolution rule for SAT (used to provide proofs for the unsatisfiable cores returned by the SAT solver), we also obtain a straightforward way to generate proofs for our MaxSAT solver — although this is not the main focus of the paper.

As our second contribution, we identify another crucial aspect of Core-guided solvers: the sequence of MaxSAT rule applications. A MaxSAT solver essentially performs a sequence of MaxSAT rule applications, related to the reformulations of the core returned by the SAT solver. Multiple such sequences are possible, and we prove that some lead to exponentially harder SAT instances. Avoiding these harder sequences is a priority. We discuss strategies to circumvent such sequences in terms of runtime. In particular, we explore the use of multiple configurations of our solver to investigate more promising sequences. A machine learning (ML) model is trained and used to select the most appropriate configuration.

In our experimental investigation, we first show that a simple MaxSAT solver can perform competitively with state-of-the-art solvers by carefully selecting the MaxSAT rule, the way we apply it to the soft constraints in the core returned by the SAT solver, and the application of the Extension rule. Second, by leveraging multiple configurations and an ML model, we effectively select a promising configuration for a given MaxSAT instance.

Our final approach outperforms the state-of-the-art. While simple, it proves effective, offering a strong foundation for future improvements and opening new avenues for certifying MaxSAT solvers.

[1]The conference version of this paper was originally submitted to the 31st International Conference on Principles and Practice of Constraint Programming (CP 2025) on March 25th, 2025.

2 Preliminaries

Definition 2.1. An arbitrary propositional formula is an expression built recursively from propositional variables x, logical connectives ($\neg,\wedge,\vee,\rightarrow,\leftrightarrow$) [2], and constants 0 (True) and 1 (False). When using the term *constraint* we will also refer to an arbitrary propositional formula. Abusing the term, we will also refer to arbitrary propositional formulas as Non-CNF formulas in this context.

Definition 2.2. A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. An empty clause, denoted as $\Box$, has no literals. A clause with just one literal is a unit clause. A Conjunctive Normal Form (CNF) is a conjunction of clauses. We will also refer to a propositional formula in CNF form as a SAT formula (or SAT instance). Abusing the notation, we can denote a CNF formula that contains an empty clause as $\Box$ since we are using this notation as a replacement for the constant False.

Definition 2.3. A weighted clause is a pair (c, w) (also noted as $_{(w)}\ c$), where c is a clause and w, its weight, is a natural number or infinity. A clause is hard if its weight is infinity (or no weight is given); otherwise, it is soft. We will denote also hard clauses (c, ∞) as $[c]$. A Weighted Partial MaxSAT instance is a multiset of weighted clauses.

Definition 2.4. A truth assignment for an instance φ is a mapping that assigns to each propositional variable in φ either 0 (False) or 1 (True). A truth assignment is *partial* if the mapping is not defined for all the propositional variables in φ.

Definition 2.5. A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. The cost of a clause (c, w) under I is 0 if I satisfies the clause; otherwise, it is w. Any truth assignment satisfies (falsifies) constant True (False). Given a partial truth assignment I, a literal or a clause is undefined if it is neither satisfied nor falsified. A clause c is a unit clause under I if c is not satisfied by I and contains exactly one undefined literal. An empty clause, $\Box$, can not be satisfied.

Definition 2.6. An unsatisfiable core is a subset of clauses of a SAT instance that is unsatisfiable.

Definition 2.7. The cost of a formula ϕ under a truth assignment I, denoted by $cost(I, \phi)$, is the aggregated cost of all its clauses under I.

Definition 2.8. The Weighted Partial MaxSAT problem for an instance ϕ is to find an assignment in which the sum of weights of the falsified soft clauses is minimal, denoted by cost(ϕ), and all the hard clauses are satisfied. The Partial MaxSAT problem is the Weighted Partial MaxSAT problem where all weights of soft clauses are equal. The SAT problem is the Partial MaxSAT problem when there are no soft clauses. An instance of Weighted Partial MaxSAT, or any of its variants, is unsatisfiable if its optimal cost is ∞. A SAT instance ϕ is satisfiable if there is a truth assignment I, called a model, such that $cost(I, \phi) = 0$.

Definition 2.9. We say that ϕ_1 and ϕ_2 are MaxSAT-equivalent if, for any truth assignment I on the variables of ϕ_1 and ϕ_2, we have that $cost(I, \phi_1) = cost(I, \phi_2)$.

Definition 2.10. A *pseudo-Boolean* (PB) constraint is a Boolean function of the form $\sum_{i=1}^{n} q_i l_i \diamond k$, where k and q_i are integer constants, l_i are literals, and $\diamond \in \{<, \leq, =, \geq, >\}$.
A Cardinality (Card) constraint is a PB constraint where all q_i are equal to 1.
An *At-Least-One* (ALO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \geq 1$.
An *At-Least-K* (ALK) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \geq k$.

Definition 2.11. Unit Propagation (UP) on a CNF formula satisfies unit clauses by assigning their sole literal to true (until fixpoint or a conflict).

[2] where $A \rightarrow B$ and $A \leftrightarrow B$ are abbreviations of $\neg A \vee B$ and $(A \rightarrow B) \wedge (B \rightarrow A)$, respectively

Definition 2.12. A SAT encoding φ of a constraint C is **consistent** if:

whenever an assignment to the varibles of the constraint C is not compatible with any solution to C, then, UP on the corresponding interpretation to the variables of φ leads to falsify some clause.

Definition 2.13. A SAT encoding φ of a constraint C is **arc-consistent** if:

- it is consistent, and
- UP discards arc-inconsistent values (i.e., values without a support in C)

Proof systems can be expressed as a set of rules, and proof complexity cares about the sizes of proofs in propositional proof systems. To compare the relative strength of different systems, the notion of *simulation* is used (Cook and Reckhow 1979).

Definition 2.14. Proof system P simulates proof system Q if every Q-proof can be converted, with at most a polynomial increase in size, into a P-proof of the same formula.

3 Non-CNF MaxSAT Rules

In this section, we present several MaxSAT rules. We use φ to denote arbitrary (Non-CNF) propositional formulas. These rules must be interpreted as replacing a set of Non-CNF propositional formulas (premises) with another set of Non-CNF propositional formulas (conclusions), such that the number of falsified premises and conclusions under any interpretation is the same. Any MaxSAT rule r can be read vertically in both directions: directly (top-down), denoted as r, and inversely (bottom-up), denoted as r^{-1}, interchanging the roles of premises and conclusions. Later, in Section 5, we will see that we can add an additional set of hard clauses to translate the conclusions into CNF, which may lead to more efficient encodings (also to more powerful proof systems).

Let us first examine the resolution rule (Davis and Putnam 1960) for the SAT problem and its Non-CNF version.

Resolution (*Res*)	Non-CNF *Res* (*R*)
Res	**R**
$x \vee A$	$\varphi \vee \varphi_1$
$\neg x \vee B$	$\neg\varphi \vee \varphi_2$
$A \vee B$	$\varphi_1 \vee \varphi_2$

where x is propostional variable and A and B are CNF clauses.

The resolution rule for SAT (*Res*) and its non-CNF version (*R*) cannot be used as MaxSAT rules because, for example, any interpretation that makes only A (φ_1) true or only B (φ_2) true falsifies one premise but not the conclusion in *Res* (*R*), and therefore they do not preserve the number of falsified premises.

From now on, we focus on the Non-CNF version of *Res*. The first observation is that, to make this rule sound for MaxSAT, we can simply add as a single conclusion the conjunction of the premises (see *MSR*). The resulting rule preserves the number of falsified premises, in contrast to *Res*. We could also replace the conclusion with a logically equivalent formula, such as $(\varphi \vee \varphi_1) \wedge (\neg\varphi \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_2)$, which corresponds to applying rule R to the conclusion (see *MSR as R*).

MaxSAT version of *R* (*MSR*)

$$\textbf{MSR}\quad \frac{\varphi \lor \varphi_1 \qquad \neg\varphi \lor \varphi_2}{(\varphi \lor \varphi_1) \land (\neg\varphi \lor \varphi_2)} \qquad\qquad \textbf{MSR as R}\quad \frac{\varphi \lor \varphi_1 \qquad \neg\varphi \lor \varphi_2}{(\varphi \lor \varphi_1) \land (\neg\varphi \lor \varphi_2) \land (\varphi_1 \lor \varphi_2)}$$

Taking rule *MSR* as our basic MaxSAT rule, we introduce some instantiations of this rule (*I0*, *I1*, and *I2*). For *I0*, φ_1 and φ_2 are false. For *I1*, φ_1 is false (notice that $\varphi \land (\neg\varphi \lor \varphi_2) \equiv \varphi \land (\neg\varphi \lor \varphi_2) \land \varphi_2 \equiv \varphi \land \varphi_2$). For *I2*, $\varphi_1 \equiv \varphi_2$. We also present some derived rules (*D3*, *D4*, and *D5*) from the application of the above instantiations of *MSR*.

$$\textbf{I0}\ \frac{\varphi \quad \neg\varphi}{\Box} \qquad \textbf{I1}\ \frac{\varphi \quad \neg\varphi \lor \varphi_2}{\varphi \land \varphi_2} \qquad \textbf{I2}\ \frac{\varphi \lor \varphi_1 \quad \neg\varphi \lor \varphi_1}{\varphi_1} \qquad \textbf{D3}\ \frac{\varphi \quad \neg\varphi \lor \varphi_2}{\varphi_2 \quad \neg\varphi_2 \lor \varphi} \qquad \textbf{D4}\ \frac{\varphi_1 \quad \varphi_2}{\varphi_1 \lor \varphi_2 \quad \varphi_1 \land \varphi_2} \qquad \textbf{D5}\ \frac{\varphi \lor \varphi_1 \quad \neg\varphi \lor \varphi_2}{\varphi_1 \lor \varphi_2 \quad \varphi \lor \varphi_1 \lor \neg\varphi_2 \quad \neg\varphi \lor \neg\varphi_1 \lor \varphi_2}$$

We want to use these rules to describe how Core-guided solvers work, i.e., rules that can be used to describe the proof system behind these Core-guided solvers and allow us to generate certificates for Core-guided solvers.

Using the notion of *simulation*, we will show for example, that by applying rules *I0* and *I1* (instantiations of *MSR*) we can *simulate I2*, *D3*, *D4* and *D5*.

Notice also that *I0* is just an instantiation of *I1* where φ_2 is false, so in some sense we could just refer to *I1*. Actually, if we look at Frege systems with Modus Ponens as its only rule of inference, we can see that its MaxSAT version, following the previous rationale, is precisely rule *I1*.

Modus Ponens (*MP*) — MaxSAT version of MP (*MSMP*)

$$\textbf{MP}\quad \frac{\varphi \qquad \neg\varphi \lor \varphi_1}{\varphi_1} \qquad\qquad \textbf{MSMP (I1)}\quad \frac{\varphi \qquad \neg\varphi \lor \varphi_1}{\varphi \land \varphi_1}$$

Lemma 3.1. *Rules I0 and I1, in combination with propositional logic equivalence rules, can simulate rule I2.*

Proof. In the following, when needed, for the sake of clarity, we specify the rule we apply on the right side of the premises, and we use the blue color to remark the conclusions the rule generates.

$$
\begin{array}{cl}
\varphi \lor \varphi_1 & \\
\neg\varphi \lor \varphi_1 & \\
\square & \text{I0}^{-1} \\
\hline
\varphi \lor \varphi_1 & \\
\neg\varphi \lor \varphi_1 & \text{I1} \\
\varphi_1 & \\
\neg\varphi_1 & \text{I1} \\
\hline
\varphi \lor \varphi_1 & \\
\varphi_1 & \\
\neg\varphi \land \neg\varphi_1 & \text{Morgan} \\
\hline
\varphi \lor \varphi_1 & \text{I0} \\
\varphi_1 & \\
\neg(\varphi \lor \varphi_1) & \text{I0} \\
\hline
\varphi_1 & \\
\square &
\end{array}
$$

Notice that we can simplify the empty clause in the original premises with the empty clause in the last conclusions, obtaining rule *I2*. □

LEMMA 3.2. *Rules I0, I1, in combination with propositional logic equivalence rules, can simulate rules D3, D4 and D5.*

PROOF. We first show that rule *D3* is obtained by applying rule *I1* twice, rule *D4* is obtained by applying rules *I2* and *I1*, and rule *D5* is obtained by applying rule *I2* three times. Notice that rule *I2* can be simulated by rules *I0* and *I1* as well (see Lemma 3.1).

Deriving *D3*

$$
\begin{array}{cl}
\varphi & \text{I1} \\
\neg\varphi \lor \varphi_2 & \text{I1} \\
\hline
\varphi \land \varphi_2 & \text{Commutative} \\
\hline
\varphi_2 \land \varphi & \text{I1}^{-1} \\
\hline
\varphi_2 & \\
\neg\varphi_2 \lor \varphi &
\end{array}
$$

Deriving *D4*

$$
\begin{array}{cl}
\varphi_1 & \\
\varphi_2 & \text{I2}^{-1} \\
\hline
\varphi_1 & \text{I1} \\
\varphi_1 \lor \varphi_2 & \\
\neg\varphi_1 \lor \varphi_2 & \text{I1} \\
\hline
\varphi_1 \lor \varphi_2 & \\
\varphi_1 \land \varphi_2 &
\end{array}
$$

Deriving *D5*

$$
\begin{array}{cl}
\varphi \lor \varphi_1 & \text{I2}^{-1} \\
\neg\varphi \lor \varphi_2 & \\
\hline
\neg\varphi \lor \varphi_2 & \text{I2}^{-1} \\
\varphi \lor \varphi_1 \lor \varphi_2 & \\
\varphi \lor \varphi_1 \lor \neg\varphi_2 & \\
\hline
\varphi \lor \varphi_1 \lor \varphi_2 & \\
\varphi \lor \varphi_1 \lor \neg\varphi_2 & \\
\neg\varphi \lor \varphi_2 \lor \varphi_1 & \text{Commutative} \\
\neg\varphi \lor \varphi_2 \lor \neg\varphi_1 & \\
\hline
\varphi \lor \varphi_1 \lor \varphi_2 & \text{I2} \\
\varphi \lor \varphi_1 \lor \neg\varphi_2 & \\
\neg\varphi \lor \varphi_2 \lor \neg\varphi_1 & \\
\neg\varphi \lor \varphi_1 \lor \varphi_2 & \text{I2} \\
\hline
\varphi \lor \varphi_1 \lor \neg\varphi_2 & \\
\neg\varphi \lor \varphi_2 \lor \neg\varphi_1 & \text{Commutative} \\
\varphi_1 \lor \varphi_2 & \\
\hline
\varphi_1 \lor \varphi_2 & \\
\varphi \lor \varphi_1 \lor \neg\varphi_2 & \\
\neg\varphi \lor \neg\varphi_1 \lor \varphi_2 &
\end{array}
$$

□

3.1 Related Work

As we will see later (in Section 5.1), rule *D3* is the *Restricted MaxRes* rule presented in (Narodytska and Bacchus 2014), rule *D4* is a particular case of the implicit MaxSAT rule (Ansótegui 2021) used in the OLL algorithm (Andres et al. 2012) (see Section 5), also used in (Bonacina, Bonet, et al. 2024) and recently in (Bonacina, Levy, et al. 2025), and rule *D5* is the Non-CNF version of the MaxSAT Resolution rule described in (Larrosa and Heras 2005) and proved to be complete in (Bonet et al. 2007). In particular, rule *D5* becomes the MaxSAT Resolution rule if we replace φ with the propositional variable x and φ_1 and φ_2 with A and B, respectively, where A and B are disjunctions of literals.

There are other ways we can derive these rules, for example, the MaxSAT Resolution rule (*MaxRes*) can be derived from rule *MSR* as described below:

MaxRes

$$\frac{x \vee A \qquad \neg x \vee B}{A \vee B \qquad x \vee A \vee \neg B \qquad \neg x \vee \neg A \vee B}$$

Deriving *MaxRes*	
$x \vee A$	MSR
$\neg x \vee B$	MSR
$(x \vee A) \wedge (\neg x \vee B)$	R
$(x \vee A) \wedge (\neg x \vee B) \wedge (A \vee B)$	Commutative
$(A \vee B) \wedge (x \vee A) \wedge (\neg x \vee B)$	$\mathrm{I1}^{-1}$
$A \vee B$ $\neg(A \vee B) \vee ((x \vee A) \wedge (\neg x \vee B))$	Morgan, Distributivity Tautology
$A \vee B$ $(x \vee A \vee \neg B) \wedge (\neg x \vee \neg A \vee B)$	MSR^{-1}
$A \vee B$ $x \vee A \vee \neg B$ $\neg x \vee \neg A \vee B$	

Other inference rules, mainly limited forms of the MaxSAT Resolution rule (*MaxRes*), have been defined to simplify the formula ((Heras and Larrosa 2006), (Larrosa and Heras 2005),(C. M. Li, Manyà, and Planes 2011)). These rules are used in Branch&Bound MaxSAT solvers such as MiniMaxSat ((Heras, Larrosa, and Oliveras 2007)), wmaxsatz ((C. M. Li, Manyà, and Planes 2011), (C. M. Li, Manyà, Mohamedou, et al. 2010)) or akmaxsat ((Kügel 2010)), to make the lower bound computation more efficient. For example, they can be applied to reformulate unsatisfiable cores detected using unit propagation on the current branch and increase the lower bound. In addition, (C. M. Li, Manyà, and Planes 2011) proposes a heuristic to determine the order of the application of the resolution rule.

3.2 Weighted Versions of the MaxSAT Rules

We have only presented the unweighted version of the above rules, since the weighted version can be simulated by applying the *Fold* rule, which ensures that all premises have equal (or uniform) weight. We will refer informally to *Fold*$^{-1}$ as *unfold*.

Fold $\qquad$ *With* $w_2 = \infty$ $\qquad$ *With* $w = 0$

$$\frac{(w_1)\ \varphi \qquad (w_2)\ \varphi}{(w_1+w_2)\ \varphi} \qquad \frac{(w)\ \varphi \qquad [\varphi]}{[\varphi]} \qquad \frac{(0)\ \varphi}{\ }$$

Remark: We can apply the inverse version of the *Fold* rule to produce a soft constraint from a hard constraint, since a hard constraint can be represented as a soft clause with weight ∞. Also, notice that any soft clause with weight 0 can be erased from our formula.

4 Core-Guided Based CNF MaxSAT Rule Presented in (Ansótegui 2021)

Core-guided MaxSAT solvers iteratively reformulate a set of soft clauses that belong to an unsatisfiable core. Since we know that the clauses in a core cannot be satisfied simultaneously, the lower bound can be trivially increased. If no more unsatisfiable cores are found, the current lower bound becomes the optimum.

Many modern state-of-the-art Core-guided MaxSAT solvers are based on the underlying idea of the OLL algorithm (Andres et al. 2012). In (Ansótegui 2021), it was observed that, in general, the OLL algorithm can be described as applying a MaxSAT rule to replace the n soft constraints (clauses, in this case) involved in an unsatisfiable core, which serve as premises, with a set of n conclusions (soft clauses) that include the empty clause as one of the conclusions. Since the empty clause appears in the conclusions, the lower bound is trivially increased. This observation can be seen as a follow-up on the comment in (Filmus et al. 2020): *"..., MaxSAT solvers do not seem to use MaxRes ..."* we discussed in the introduction of the paper.

Without loss of generality[3], we now assume that our soft clauses are unit clauses of the form x_i. In (Ansótegui 2021), in an attempt to describe Core-guided MaxSAT solvers (implementing the OLL algorithm) from the perspective of MaxSAT rules, the *Core-guided based MaxSAT* rule was introduced along with the *At Most One* rule for preprocessing introduced in (Ignatiev, Morgado, et al. 2019).

Core-guided based MaxSAT rule in (Ansótegui 2021)

$$\frac{\begin{array}{c} x_1 \\ \cdots \\ x_n \end{array}}{\begin{array}{c} x'_1 \\ \cdots \\ x'_{n-1} \\ \square \\ [\mathrm{AL}(i : x_1, \ldots, x_n, o : x'_1, \ldots, x'_{n-1})] \end{array}}$$

where $x_1 + \cdots + x_n < n$

At Most One MaxSAT rule in (Ignatiev, Morgado, et al. 2019)

$$\frac{\begin{array}{c} x_1 \\ \cdots \\ x_n \end{array}}{\begin{array}{c} x_1 \vee \cdots \vee x_n \\ {}_{(n-1)}\ \square \end{array}}$$

where $x_1 + \cdots + x_n \leq 1$

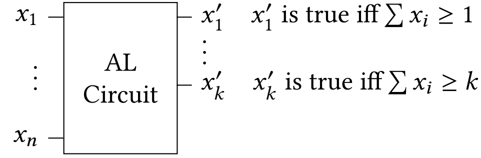
Here, we introduce a more general rule called *Cardinality-guided MaxSAT* rule that covers the two previous cases. The condition for applying this rule is that $x_1 + \cdots + x_n \leq k$. Setting $k = n - 1$ covers the case where the premises belong to an unsatisfiable core, while $k = 1$ corresponds to the AMO (At Most One) technique used in MaxSAT preprocessing, as described in (Ignatiev, Morgado, et al. 2019).

[3] Any MaxSAT formula can be translated into a MaxSAT-equivalent formula in which soft clauses are unit clauses.

Cardinality-guided MaxSAT rule

$$\frac{\begin{array}{c} x_1 \\ \dots \\ x_n \end{array}}{\begin{array}{c} x'_1 \\ \dots \\ x'_k \\ (n-k)\ \square \\ [\mathrm{AL}(i : x_1, \dots, x_n, o : x'_1, \dots, x'_k)] \end{array}}$$

where $x_1 + \cdots + x_n \leq k,\ 0 \leq k \leq n$



The premises in this rule are a set of n soft unit clauses x_i in a MaxSAT formula ϕ, which will be replaced by the following conclusions: a set of k soft unit clauses x'_i, $n-k$ empty clauses, and $\mathrm{AL}(i : x_1, \dots, x_k,\ o : x'_1, \dots, x'_k)$, a set of hard clauses representing a circuit that takes the premises $x_1, \dots, x_n$ as input and produces the conclusions $x'_1, \dots, x'_k$ as output.

The AL circuit guarantees that x'_i is true iff $\sum_{j=1}^{n} x_j \geq i$. The circuit is translated into CNF and added as a set of hard clauses. This circuit can be interpreted as an At Least (AL) cardinality constraint.

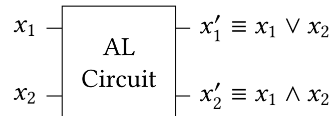
Lemma 4.1. *Cardinality-guided MaxSAT rule preserves MaxSAT equivalence.*

Proof. We need to show that for any interpretation that satisfies the condition $x_1 + \cdots + x_n \leq k,\ 0 \leq k \leq n$, if m premises are falsified, then exactly m conclusions are falsified.

Any interpretation satisfying $x_1 + \cdots + x_n \leq k$, has to falsify at least $n-k$ x_i literals, which actually corresponds to the $n-k$ output empty clauses.

Since x'_i is true iff $\sum_{j=1}^{n} x_j \geq i$, any output literal x'_i with index $i > n-m$ (a total of $m-(n-k)$ literals x'_i) has to be false and the rest true. □

If we consider the case $k = n = 2$ (see figure below), with inputs x_1 and x_2, the circuit can trivially encode the constraints $x_1 + x_2 \geq 1$ and $x_1 + x_2 \geq 2$ as $x_1 \lor x_2$ and $x_1 \land x_2$, respectively. This is precisely the *structure* of the MaxSAT rule *D4*.



5 Cardinality-Guided Based Non-CNF MaxSAT Rules

Let us now take a step back to our more general formalism, where soft constraints are arbitrary propositional formulas (i.e., Non-CNF formulas) rather than just CNF clauses. We can describe the previous *Cardinality-guided MaxSAT* rule as a combination of three components: the new *Non-CNF-AL* MaxSAT rule, *Fold*$^{-1}$, and *I0*, as follows:

Non-CNF-AL

$$\begin{array}{c} \varphi_1 \\ \dots \\ \varphi_n \\ \hline \varphi'_1 \\ \dots \\ \varphi'_n \end{array}$$

where $\varphi'_i \equiv \sum_{j=1}^{n} \varphi_j \geq i$

Fold-1

$$\begin{array}{c} [\neg\varphi'_i] \\ \hline \neg\varphi'_i \\ [\neg\varphi'_i] \end{array}$$

I0

$$\begin{array}{c} \varphi'_i \\ \neg\varphi'_i \\ \hline \square \end{array}$$

Cardinality-guided Non-CNF-AL

(Non-CNF-AL, Fold-1, I0)

$$\begin{array}{c} \varphi_1 \\ \dots \\ \varphi_n \\ \hline \varphi'_1 \\ \dots \\ \varphi'_k \\ (n-k)\ \square \end{array}$$

where $\varphi_1 + \cdots + \varphi_n \leq k,\ 0 \leq k \leq n$

The *Non-CNF-AL* MaxSAT rule replaces a set of n soft constraints φ_i with another set of n soft constraints φ'_i, such that $\varphi'_i \equiv \sum_{j=1}^{n} \varphi_j \geq i$. This can be interpreted as an At Least (AL) Non-CNF circuit.

If we know that the set of premises satisfies the condition $\varphi_1 + \cdots + \varphi_n \leq k$, where $0 \leq k \leq n$, then we also know that no more than k of the original constraints can be satisfied simultaneously—i.e., at least $n - k$ of them must be false. In other words, $\neg\varphi'_i$ is true for all $i > k$.

Therefore, we can explicitly extend the set of hard constraints with $\neg\varphi'_i$ for $i > k$, and then apply the inverse version of the *Fold* rule to obtain these as soft constraints. Finally, we apply the MaxSAT rule *I0* to replace each pair φ_i and $\neg\varphi'_i$ with $\square$ for $i > k$, trivially increasing the lower bound by $n - k$.

Lemma 5.1. *Cardinality-guided Non-CNF-AL rule preserves MaxSAT equivalence.*

Proof. The proof is similar to proof in Lemma 4.1 with the exception that the role of the AL circuit is captured by the semantics of φ'_i formulas. □

5.1 Simulating *Non-CNF-AL* Rule with *Simpler* MaxSAT Rules

Now, we focus our attention on the *Non-CNF-AL* rule described in the previous section, considering the case with just two soft constraints as premises. One of the simplest ways to explicitly define the conclusions (φ'_1 and φ'_2) is to replace them with $\varphi_1 \lor \varphi_2$ and $\varphi_1 \land \varphi_2$, respectively. As mentioned earlier, this corresponds to the MaxSAT rule *D4*, presented in Section 3.

Non-CNF-AL $\approx$ **D4**

$$\begin{array}{c} \varphi_1 \\ \varphi_2 \\ \hline \varphi'_1 \\ \varphi'_2 \end{array} \quad \approx \quad \begin{array}{c} \varphi_1 \\ \varphi_2 \\ \hline \varphi_1 \lor \varphi_2 \\ \varphi_1 \land \varphi_2 \end{array}$$

where $\varphi'_1 \equiv \varphi_1 + \varphi_2 \geq 1$,
$\varphi'_2 \equiv \varphi_1 + \varphi_2 \geq 2$

Noticing that for the case of two premises, *Non-CNF AL* can be expressed as a single application of *D4* rule, we further investigate if there is a sequence of applications of rule *D4* that simulate *Non-CNF AL*. To this end, we introduce two algorithms that implement a strategy to apply rule *D4*.

In Algorithm ALK_1, given a list of input constraints, we generate a list of output constraints where the last output constraint plays the role of the highest comparator in an At-Least cardinality constraint, i.e., the sum of

satisfied inputs must be at least the number of inputs. Specifically, from a set of input constraints $\varphi_1, \ldots, \varphi_n$, we produce the following set of conclusions:

$$\bigwedge_{i=1}^{i=n-1} \left((\bigwedge_{j=1}^{j=i} \varphi_j) \vee \varphi_{i+1} \right)$$

and the last conclusion is $(\varphi_1 \wedge \cdots \wedge \varphi_k)$, which semantically represents that all input constraints have to be True.

The functions `popfirst`, `append`, and `prepend` extract the first element, append an element to the end, and append an element to the front of the list, respectively. In essence, we traverse the list of input constraints from left to right: we replace the first two constraints in the list with the AND conclusion of rule *D4* and append the OR conclusion to the output. The last AND conclusion, which is appended to the output, corresponds to the highest comparator, as it is the AND of all input constraints.

Algorithm ALK$_1$:

```
Input  : I: List of input constraints (at least two).
Output : O: List of output constraints.
1  while len(I) > 1 do
2      φ1 ← I.popfirst()
3      φ2 ← I.popfirst()
4      φ'_OR, φ'_AND ← D4(φ1, φ2)
5      O.append(φ'_OR)
6      if len(I) ≥ 1 then
7          I.prepend(φ'_AND)
8      else
9          O.append(φ'_AND)
10 return O
```

Algorithm ALK$_k$:

```
Input  : I: List of input constraints (at least two). k: number of comparators (with k <= len(I)).
Output : O: List of output constraints.
1  for i ∈ [1, k] do
2      I = ALK1(I)
3      O.prepend(I.poplast())
4      if len(I) == 1 then
5          O.prepend(I.poplast())
6          return concatenate(I, O)
7  return concatenate(I, O)
```

In Algorithm ALK$_k$, we iterate Algorithm ALK$_1$ k times to produce k output constraints [4], which play the role of comparators in such a way that, if we set the i-th output constraint (starting at 1) to true, then at least i input constraints must be true. The function `poplast` extracts the last element of the list, and the function `concatenate` concatenates the second input list to the first.

Notice that we are applying a MaxSAT rule at every step, so any intermediate output of the process is a set of constraints that can be used to replace the input constraints from the perspective of MaxSAT.

Example 5.2. Let $\varphi_1, \varphi_2, \varphi_3$ be the input constraints to Algorithm ALK$_k$ with $k = 3$. The sequence of applications of rule *D4* is as follows:

[4] If $k = n$, to produce k output constraints playing the role of comparators we only need $k - 1$ calls

First call to Algorithm ALK_1	
φ_1	D4
φ_2	D4
φ_3	
$\varphi_1 \vee \varphi_2$	
$\varphi_1 \wedge \varphi_2$	D4
φ_3	D4
$\varphi_1 \vee \varphi_2$	
$\varphi_1 \wedge \varphi_2 \vee \varphi_3$	
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	

Second call to Algorithm ALK_1	
$\varphi_1 \vee \varphi_2$	D4
$\varphi_1 \wedge \varphi_2 \vee \varphi_3$	D4
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	
$(\varphi_1 \vee \varphi_2) \vee (\varphi_1 \wedge \varphi_2 \vee \varphi_3)$	
$(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \wedge \varphi_2 \vee \varphi_3)$	
$\varphi_1 \wedge \varphi_2 \wedge \varphi_3$	

The last three conclusions are, in order of appearance, logically equivalent to: $(\varphi_1 \vee \varphi_2 \vee \varphi_3)$, $(\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3) \wedge (\varphi_2 \vee \varphi_3)$, and $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$, respectively. These correspond semantically to comparators ≥ 1, ≥ 2, and ≥ 3.

Lemma 5.3. *Rule D4 can simulate Non-CNF-AL MaxSAT rule on n premises.*

Proof. By construction, notice that ALK_k with $(k = n)$ produces all n comparators with the semantics in *Non-CNF-AL* MaxSAT rule. □

Lemma 5.4. *Rules I0 and I1, can simulate Non-CNF-AL rule on n premises.*

Proof. Notice that *I0* and *I1* can simulate *D4* (see Lemma 3.2), and *D4* can simulate *Non-CNF-AL* (see Lemma 5.3). □

As a side note on related work, the *Restricted MaxRes* rule used in (Narodytska and Bacchus 2014) can be applied to derive the same conclusions as those produced in the first call to ALK_1 in Example 5.2. Although not described this way in the original paper, essentially, the way *Restricted MaxRes* is used in PMRes algorithm corresponds to rule *D3*, along with the implicit application of rule *I0*, which is triggered with the core returned by the SAT solver. Notice that PMRes can only emulate ALK_k if the SAT solver provides k cores consecutively on the recurrent outputs of ALK_1 (in the terminology of this paper, PMRes can only provide one comparator when reformulating the soft constraints in the core, while with ALK_k we can produce k comparators).

Example 5.5. We apply rules *I0* and *D3* to the formula in Example 5.2. First, we will assume that we add $\square$ to the premises, which we will recover later.

φ_3	
φ_2	
φ_1	
$\square$	$\mathrm{I0}^{-1}$
φ_3	
φ_2	
φ_1	
$\neg(\varphi_3 \wedge \varphi_2 \wedge \varphi_1)$	Morgan
$\varphi_3 \wedge \varphi_2 \wedge \varphi_1$	
φ_3	D3,Morgan
φ_2	
φ_1	
$\neg\varphi_3 \vee \neg\varphi_2 \vee \neg\varphi_1$	D3,Morgan
$\varphi_3 \wedge \varphi_2 \wedge \varphi_1$	
φ_2	D3,Morgan
φ_1	
$\neg\varphi_2 \vee \neg\varphi_1$	D3,Morgan
$\varphi_2 \wedge \varphi_1 \vee \varphi_3$	
$\varphi_3 \wedge \varphi_2 \wedge \varphi_1$	
φ_1	*I*0
$\neg\varphi_1$	*I*0
$\varphi_1 \vee \varphi_2$	
$\varphi_2 \wedge \varphi_1 \vee \varphi_3$	
$\varphi_3 \wedge \varphi_2 \wedge \varphi_1$	
$\varphi_1 \vee \varphi_2$	
$\varphi_2 \wedge \varphi_1 \vee \varphi_3$	
$\varphi_3 \wedge \varphi_2 \wedge \varphi_1$	
$\square$	

5.2 Translating into CNF with the *Extension* Rule

In the current context of MaxSAT solvers, the input soft and hard constraints must be clauses. Hard constraints are simply translated into CNF. Soft constraints are reified into auxiliary variables, replacing each constraint, say φ, with a soft unary clause containing a new fresh auxiliary variable, say x, and the hard clauses that translate into CNF $x \leftrightarrow \varphi$. We will refer to this as an application of the *Extension* rule (Tseitin 1983).

Notice that, in the context of MaxSAT solvers, since we are maximizing, we can omit the left direction $\leftarrow$ of the double implication when reifying the soft clauses. This is typically applied in modern MaxSAT solvers, although in this work we preserve both directions, since it seems to be more effective in practice.

Extension rule

$$\frac{\varphi}{\begin{array}{c} x \\ [x \leftrightarrow \varphi] \end{array}}$$

The MaxSAT rules that can simulate the *Non-CNF-AL* rule, as already mentioned, essentially generate an At-Least circuit. We will see how to translate this cardinality constraint into CNF by applying the *Extension* rule.

In the literature, many methods exist to encode cardinality constraints in SAT: the totalizer (Bailleux and Boufkhad 2003), sorting network (Batcher 1968), cardinality network (Asín et al. 2009), and sequential counter (Sinz 2005); and each of them is arc-consistent. Depending on our particular choice, the resulting SAT encoding can be more efficient in practice.

The observation that using a MaxSAT rule iteratively plus the *Extension* rule results in encoding a cardinality constraint into CNF was analyzed in (Ansótegui and Gabàs 2017). In that paper, it is shown that the PMRes algorithm (Narodytska and Bacchus 2014) iteratively builds, implicitly, a sequential counter.

Example 5.6. We now illustrate how we translate into CNF (using the *Extension* rule) the output conclusions from Algorithm ALK$_1$ on four input soft constraints, $\varphi_1, \varphi_2, \varphi_3, \varphi_4$, in three different ways.

For all three approaches, we first reify the input soft constraints into the corresponding x_i variables. For `ext2`, we always reify the conclusions of *D4* into a new x_i variable that we will use as input premise for the next applications of *D4* rule. For `ext1`, we do as for `ext2` but reification variables that semantically represent the conjunction of input premises, when operated in a new conjunction, are replaced by the conjunction they represent, i.e, with no intermediate additional reification variables. For `ext3` we follow the same reification strategy as in (Narodytska and Bacchus 2014). Notice that in practice, if the input soft constraints can not be satisfied simultaneously, clause x_{10} can vanish from `ext1` and `ext2`, and clause x_{11} will vanish from `ext3`. Moreover, in `ext3`, we can rename variable x_5 as x_4.

ALK$_1$ in CNF (ext1)	ALK$_1$ in CNF (ext2)	ALK$_1$ in CNF (ext3) as in PMRes (Narodytska and Bacchus 2014)
φ_1	φ_1	φ_4
φ_2	φ_2	φ_3
φ_3	φ_3	φ_2
φ_4	φ_4	φ_1
x_5	x_5	x_8
x_7	x_7	x_9
x_9	x_9	x_{10}
x_{10}	x_{10}	x_{11}
$[x_1 \leftrightarrow \varphi_1]$	$[x_1 \leftrightarrow \varphi_1]$	$[x_1 \leftrightarrow \varphi_1]$
$[x_2 \leftrightarrow \varphi_2]$	$[x_2 \leftrightarrow \varphi_2]$	$[x_2 \leftrightarrow \varphi_2]$
$[x_3 \leftrightarrow \varphi_3]$	$[x_3 \leftrightarrow \varphi_3]$	$[x_3 \leftrightarrow \varphi_3]$
$[x_4 \leftrightarrow \varphi_4]$	$[x_4 \leftrightarrow \varphi_4]$	$[x_4 \leftrightarrow \varphi_4]$
$[x_5 \leftrightarrow x_1 \lor x_2]$	$[x_5 \leftrightarrow x_1 \lor x_2]$	$[x_5 \leftrightarrow x_4]$
$[x_6 \leftrightarrow x_1 \land x_2]$	$[x_6 \leftrightarrow x_1 \land x_2]$	$[x_6 \leftrightarrow x_3 \land x_5]$
$[x_7 \leftrightarrow x_6 \lor x_3]$	$[x_7 \leftrightarrow x_6 \lor x_3]$	$[x_7 \leftrightarrow x_2 \land x_6]$
$[x_8 \leftrightarrow x_1 \land x_2 \land x_3]$	$[x_8 \leftrightarrow x_6 \land x_3]$	$[x_{11} \leftrightarrow x_1 \land x_7]$
$[x_9 \leftrightarrow x_8 \lor x_4]$	$[x_9 \leftrightarrow x_8 \lor x_4]$	$[x_8 \leftrightarrow x_1 \lor x_7]$
$[x_{10} \leftrightarrow x_1 \land x_2 \land x_3 \land x_4]$	$[x_{10} \leftrightarrow x_8 \land x_4]$	$[x_9 \leftrightarrow x_2 \lor x_6]$
		$[x_{10} \leftrightarrow x_3 \lor x_5]$

In our experiments, strategy `ext2` was the most performant.

Remark: As already mentioned, we can apply rule *D4* in a different order to the input soft constraints to obtain other encodings of cardinality constraints, such as a totalizer (Bailleux and Boufkhad 2003). However, we reiterate that it is crucial to carefully design how we apply the *Extension* rule and how we interconnect auxiliary variables.

Lemma 5.7. *The ALK_1 procedure on m unit clauses and $ext2$ strategy for Extension rule, generates $2 \cdot (m-1)$ ternary clauses, $4 \cdot (m-1)$ binary clauses.*

Proof. Notice that ALK_1 applies exactly $m-1$ times MaxSAT rule *D4*. Rule *D4* produces 2 conclusions involving two literals each. *Extension* rule as in $ext2$ produces 1 additional ternary clause and two binary clauses for each conclusion of 2 literals. □

Lemma 5.8. *The ALK_n procedure on m soft unit clauses and $ext2$ strategy for Extension rule, produces $2((n-1) \cdot m - (n \cdot (n-1)/2))$ ternary clauses and $4((n-1) \cdot m - (n \cdot (n-1)/2))$ binary clauses.*

Proof. Notice that ALK_n calls exactly $n-1$ times to algorithm ALK_1. These gives us $(n-1) \cdot m - (n \cdot (n-1)/2)$ applications of *D4* MaxSAT rule. □

To summarize this section, modern Core-guided solvers can be described as essentially applying a MaxSAT rule iteratively to a set of input constraints that can not be satisfied simultaneously, in such a way that one of the conclusions becomes the empty clause □, allowing us to increase the lower bound. Since not all MaxSAT rules produce conclusions in CNF form, the *Extension* rule is applied. Depending on the order in which we apply the MaxSAT rule to the set of constraints in the core (e.g. Algorithm ALK_k), how the *Extension* rule is applied (e.g. strategy `ext2`), and even how we add further constraints involving the auxiliary variables from the *Extension* rule (e.g. to improve the level of arc-consistency of the encoding), different reformulation approaches for the input constraints are obtained, which directly impact the efficiency of the MaxSAT solver.

Notice that nothing prevents us from applying the MaxSAT rule to a set of input constraints that do not form a core. However, at some point, we need to reach a reformulation from which we can derive empty clauses easily and then increase the lower bound, or at least be able to estimate or compute the lower bound using other techniques.

6 A Complete MaxSAT Algorithm

In this section, we provide the basic schema of a MaxSAT algorithm, which we name MSAT. We assume the following global variables: $softs$, which contains the soft constraints of our problem once translated into unary clauses; F, which represents the current number of empty clauses; ub_model, which stores the current best model; and S, the incremental SAT solver that we iteratively call. We also consider k as a fixed parameter of the algorithm representing the number of rounds for the Algorithm ALK_k.

In Algorithm MSAT, we: (1) initialize the incremental SAT solver, (2) load the input MaxSAT instance such that hard clauses are added directly to the SAT solver and soft clauses are stored in $softs$ as unary clauses (clauses of length ≥ 2 are substituted by a fresh variable using the *Extension* rule) along with their weight, and the number of empty clauses (F) is initialized; and (3) compute an initial model satisfying the hard clauses.

Then, we call Algorithm MaxSAT. At each iteration, we first send to the SAT solver the soft clauses in $softs$ with a positive weight (in particular, the corresponding literals representing these soft clauses) as assumptions. The order in which we send the assumptions to the SAT solver matters, as shown in Section 7 (Figure 3), since it affects which core, among all the possible ones, the SAT solver returns. The assumptions are ordered in increasing chronological order according to their generation. For assumptions related to the original input soft clauses, we assume that soft clauses appearing earlier in the input MaxSAT instance are generated earlier.

If the solver's response ($status$) is False (i.e., the current SAT formula cannot be satisfied under these assumptions), the variable $core$ contains a core (a subset of the assumptions that makes the SAT formula unsatisfiable). We then apply any technique to refine the core (such as trimming, core minimization (Ignatiev, Morgado, et al. 2019),

etc.). If the core is empty, it means that the hard clauses cannot be satisfied, and we return UNSAT. Otherwise, we begin the process of reformulating this core to increase the current lower bound.

First, we apply the inverse of the *Fold* rule (`unfold`) until the weight of all premises is equal to the minimum weight found in the core. In practice, we subtract the minimum weight (w_{min}) from the weight of the literals of structure *softs* that are in the current core. The literals with weight w_{min} are virtually preserved apart in the core (not reintroduced into *softs*) since we postpone substituting them (see Algorithm Substitute) with a new set of literals that will be added to *softs* (lines 19 to 25). By postponing the substitution, we can detect disjoint cores (Marques-Silva and Planes 2008).

If the current number of empty clauses, plus the sum of the weights of the cores we have yet to process, is greater than or equal to the cost of the current model (i.e., the upper bound), we can stop and return the current model.

If the solver's response ($status$) is True, we compute the cost of the model returned by the SAT solver and update (`update_ub`) ub_model if this cost is lower than the cost of ub_model. If there are no more cores to process ($len(R) == 0$), the current ub_model is optimal, and we return it.

Then, we process the list of cores that need to be reformulated. We filter out those cores that, under certain criteria, we consider low-quality cores (in some sense those that can lead to harder sequences of cores, as described in the next section). Then, we call Algorithm Substitute, which returns a new set of soft clauses. We then perform cover optimization (Ansótegui, Gabàs, and Levy 2016) by calling the MaxSAT function again, but only on the new soft clauses. The goal is to potentially limit the number of inputs that can be true simultaneously by finding an additional core among the new soft clauses.

Finally, if at any iteration the current number of empty clauses is greater than or equal to the current upper bound, we stop and return the current ub_model.

Algorithm MSAT:

Input : ϕ: MaxSAT formula
Output: Returns UNSAT if hard constraints can not be satisfied, otherwise an optimal solution for ϕ.

1 $S \leftarrow$ SAT()
2 $softs, F \leftarrow$ load_MaxSAT_instance(ϕ, S)
3 $ub_model \leftarrow$ initialize_UB()
4 **return** *MaxSAT(softs)*

In Algorithm Substitute, we replace a set of soft clauses that form a core for another set of soft clauses. First, we call the `ALK_n` function (which, in this case, generates k output comparators). `ALK_n` returns a new set of literals representing the new soft clauses, along with a set of hard clauses. These hard clauses come from the application of the *Extension* rule, as well as any additional clauses introduced for efficiency in the encoding (as described in Section 5.1). These hard clauses are then added to the SAT solver.

We also add a hard clause representing the negation of the literal in $outputLits$ that corresponds to the conjunction of all the soft clauses in the core. Additionally, we increase the number of empty clauses by the weight shared by all soft constraints in the core.

Finally, we return the new set of soft clauses.

Algorithm MaxSAT:

```
   Input  : softs: soft unary clauses
   Output : Returns UNSAT if hard constraints can not be satisfied, otherwise an optimal solution for ϕ.
 1 R ← [ ] // List of cores to reformulate
 2 while True do
 3     status, core, model ← S.solve(softs with weight > 0)
 4     if status is False then
 5         // Optionally, refine the core here (trimming, minimization, etc)
 6         if len(core) == 0 then
 7             return UNSAT
 8         // Prepare core for Reformulation
 9         w_min ← unfold(core)
10         R.append(⟨ core, w_min ⟩)
11         if F + ∑ w ∈ R ≥ cost(ub_model, ϕ) then
12             return ub_model
13         continue
14     if status is True then
15         ub_model ← update_ub(model)
16         if len(R) == 0 then
17             return ub_model
18     // Core Reformulation:
19     while len(R) > 0 do
20         inputLits, w ← R.popfirst()
21         if filter condition then
22             continue
23         outputLits ← substitute(inputLits, w)
24         // Cover optimization
25         MaxSAT(outputLits)
26     if F ≥ cost(ub_model, ϕ) then
27         return ub_model
```

6.1 Generating Certificates for Algorithm MaxSAT

By describing and implementing the Algorithm MaxSAT with MaxSAT rules, we can easily produce a proof (certificate) for a MaxSAT instance.

Modern SAT solvers can provide a proof for the core they return, which can be described as a sequence of applications of the *Res* rule, that can be used to create a proof of the negation of the conjunction of the soft constraints involved into the core, say $\neg\varphi$.

Algorithm Substitute:

Input : *inputLits*: List of literals representing the soft clauses to substitute (assuming there is a core between them)
w: Minimum common weight from the literals to substitute

Output : Returns the list of literals representing the generated soft clauses

```
1 outputLits, hards ← ALK_n(inputLits, k)
2 // Add the new cardinality constraint
3 S.add_clauses(hards)
4 x = outputLits.poplast()
5 S.add_clause([¬x])
6 // Add the new softs clauses
7 Add literals in outputLits to softs with weight w.
8 F ← F + w
9 return outputLits
```

Then, we concatenate to the proof of $\neg\varphi$ the sequence of applications of the *D4* and *Extension* rules, that end up producing φ (the conjunction of the soft constraints in the core as in Example 5.2). Finally, *I0* rule produces an empty clause $\square$ from $\neg\varphi$ and φ. We repeat this process for every core found by the SAT solver. If the formula is weighted, we first apply the *Fold*$^{-1}$ rule to unfold each soft constraint with weight w into two copies, one with the minimum weight of the soft constraints in the core, say w_{min}, and another with $w - w_{min}$. And, as described earlier, we concatenate the proof for the empty clause, in this case $_{(w)}\,\square$, taking as input soft constraints the copies with weight w_{min}.

Note: since *I1* (with its particular case *I0*) can simulate *D4* and *MSR*, and *MSR* rule on hard clauses is in fact the *Res* rule (see *MSR as R* in Section 3), we can produce proofs for the Algorithm MaxSAT with *I1*, *Extension*, and *Fold* rules.

7 Avoiding Hard Sequences

Traditionally, Core-guided solvers have focused on how to efficiently reformulate the default sequence of unsatisfiable cores provided by the SAT solver to replace the soft constraints. For example, given an unsatisfiable core, MaxSAT solvers have been optimized using trimming and minimization techniques (Ignatiev, Morgado, et al. 2019) to refine the core, efficient encodings for the cardinality constraints imposed on the soft constraints in the core, and alternative ways of applying cover optimization (Ansótegui, Gabàs, and Levy 2016) techniques.

However, less effort has been devoted to addressing two important problems that critically impact the efficiency of the solver. First, the number of alternative unsatisfiable cores that a SAT solver can return may be exponential. Second, the core we reformulate affects the hardness of subsequent SAT calls in such a way that different sequences of reformulated cores can result in an exponential difference in runtime.

In fact, we face a search problem in the tree of possible sequences/paths of reformulated cores. Rephrasing the popular idiom, *we can't see the tree for the default sequence/path.* Figure 1 shows an abstract view of the search space of different sequences.

It seems that a promising avenue would be to explore other possible sequences or paths. Since the branching factor of this tree is so high, we currently aim to use greedy techniques, such as restarts (discarding part of the reformulation process in a sequence), to explore alternative avenues based on some heuristics.

7.1 Exponentially Harder Core-Sequences

Let's prove the existence of alternative harder sequences. For this, we first define the Pigeon-Hole Principle, its MaxSAT version, and the time complexity of Algorithm MaxSAT on a particular case.

Definition 7.1. Let PHP^n_m denote the Pigeon-Hole Principle with n pigeons and m holes, where $n > m$. A natural way to translate this problem into SAT is to use $n \cdot m$ propositional variables $x_{i,j}$, where $x_{i,j}$ is true if pigeon i is placed in hole j.
A set of At-Least-One (ALO) constraints:

$$\bigwedge_{i=1}^{n} \sum_{j=1}^{m} x_{i,j} \geq 1$$

A set of At-Most-One (AMO) constraints:

$$\bigwedge_{j=1}^{m} \sum_{i=1}^{n} x_{i,j} \leq 1$$

The ALO and AMO constraints are encoded into CNF with an arc-consistent encoding.
In the MaxSAT version of PHP^n_m, the ALO constraints are soft, and the AMO constraints are hard.

Lemma 7.2. *Algorithm MSAT can solve the MaxSAT version of PHP^n_1 in $O(n^2)$ time.*

Proof. In essence, in the MaxSAT version of PHP^n_1 we have an At Most One constraint on the soft unary clauses, $x_1 + \cdots + x_n \leq 1$.

Let's assume for the moment that n is even. The Algorithm MSAT performs a kind of binary search to solve the MaxSAT version of PHP^n_1.

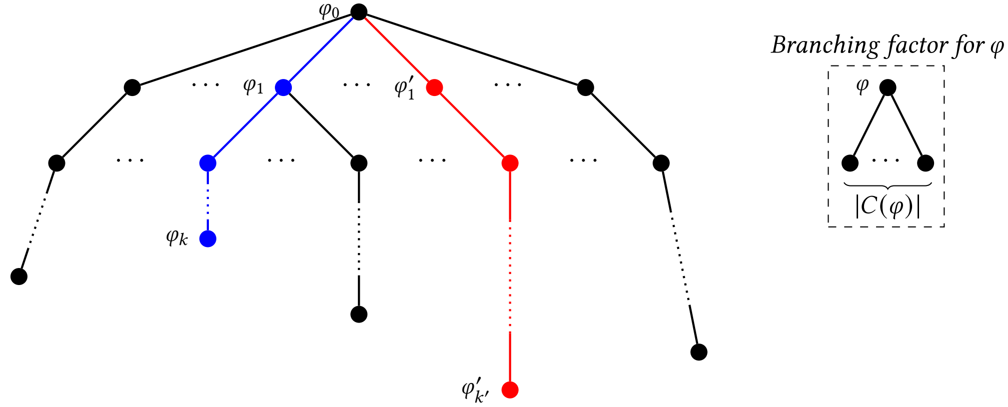
First, it finds $n/2$ cores of size 2, say (x_i, x_j), since no two input soft clauses can be true. On each of these cores, it will apply MaxSAT rule *D4*, replacing x_i and x_j with two new unit clauses on fresh variables that represent $x_i \vee x_j$ and $x_i \wedge x_j$. The conjunction is trivially false. Therefore, we have $n/2$ soft unary clauses representing the disjunction of two pairs of input soft unary clauses and $n/2$ empty clauses.

We can again apply the same strategy on the remaining $n/2$ soft unary clauses since any pair also forms a core, until we get to the point that we have just two soft unit clauses representing each the disjunction of half of the input soft unit clauses. We apply rule *D4* on the last two soft unit clauses, and we get a soft unary clause representing the disjunction of all input soft unary clauses, plus one empty clause that is added to the previously accumulated $n-2$ empty clauses, i.e., a total of $n-1$ empty clauses. Figure 2 shows a real execution with Algorithm MSAT for solving the MaxSAT version of PHP^8_1, where $x_1, \cdots, x_8$ is the set of soft clauses.

Now we need to analyze how costly it is for the solver to find those cores. Imagine we are in the penultimate step, having two unary clauses representing, for example, $x_1 \vee \cdots \vee x_{n/2}$ and $x_{n/2+1} \vee \cdots \vee x_n$. The SAT solver only needs to check for every literal in the first clause whether it is compatible with every literal in the second clause to report a core. That gives a total of $n/2 \cdot n/2$ checks for the penultimate step. The total number of checks at step k (with $k \geq 1$, where $k = 1$ is the penultimate step, $k = 2$ the antepenultimate step, etc) is $2^{k-1} * (n/2^k)^2 = n^2/2^{k+1}$. If we sum up the checks for all steps, we get $n^2 * (\sum_{k=1}^{\log_2 n} 1/2^{k+1})$ checks, which is $O(n^2)$.

□

Theorem 7.3. *Let φ_i be the SAT instance generated by Algorithm MSAT at the ith step. There exist MaxSAT formulas for which Algorithm MSAT can generate alternative sequences $\varphi_0, \varphi_1, \ldots, \varphi_k$ and $\varphi_0, \varphi'_1, \ldots, \varphi'_{k'}$, where k' is at most polynomially larger than k, and the sequence $\varphi'_1, \ldots, \varphi'_{k'}$ is exponentially harder for resolution than $\varphi_1, \ldots, \varphi_k$.*



$$Res(\varphi_0, \varphi'_1, \dots, \varphi'_{k'}) \geq_{exp} Res(\varphi_0, \varphi_1, \dots, \varphi_k)$$

Fig. 1. Example of possible sequences of SAT instances generated by Algorithm MaxSAT, where φ_i is the SAT instance generated at the ith step of the algorithm. We denote by $C(\varphi)$ the set of different unsatisfiable cores in φ. The branching factor for a node labelled with φ is $|C(\varphi)|$.

Proof. See Figure 1 for a possible tree search space of sequences of different hardness.

We begin the proof with an introductory example.

Example 7.4. Let the following working formula be given, where x_i are propositional variables, and there are 12 soft clauses x_i. The expression $[\sum x_i \leq 1]$ represents an arc-consistent CNF encoding of the At-Most-One constraint as hard clauses. The optimal cost of this instance is 9.

MaxSAT instance	After sequence 1	After sequence 2
$x_1, x_2, x_3,$	$x_1 \lor x_2 \lor x_3$	$x_1 \lor x_4 \lor x_7 \lor x_{10}$
$x_4, x_5, x_6,$	$x_4 \lor x_5 \lor x_6$	$x_2 \lor x_5 \lor x_8 \lor x_{11}$
$x_7, x_8, x_9,$	$x_7 \lor x_8 \lor x_9$	$x_3 \lor x_6 \lor x_9 \lor x_{12}$
$x_{10}, x_{11}, x_{12},$	$x_{10} \lor x_{11} \lor x_{12}$	$_{(9)}\ \square$
$[x_1 + x_2 + x_3 \leq 1]$	$_{(8)}\ \square$	$[x_1 + x_2 + x_3 \leq 1]$
$[x_4 + x_5 + x_6 \leq 1]$	$[x_1 + x_2 + x_3 \leq 1]$	$[x_4 + x_5 + x_6 \leq 1]$
$[x_7 + x_8 + x_9 \leq 1]$	$[x_4 + x_5 + x_6 \leq 1]$	$[x_7 + x_8 + x_9 \leq 1]$
$[x_{10} + x_{11} + x_{12} \leq 1]$	$[x_7 + x_8 + x_9 \leq 1]$	$[x_{10} + x_{11} + x_{12} \leq 1]$
$[x_1 + x_4 + x_7 + x_{10} \leq 1]$	$[x_{10} + x_{11} + x_{12} \leq 1]$	$[x_1 + x_4 + x_7 + x_{10} \leq 1]$
$[x_2 + x_5 + x_8 + x_{11} \leq 1]$	$[x_1 + x_4 + x_7 + x_{10} \leq 1]$	$[x_2 + x_5 + x_8 + x_{11} \leq 1]$
$[x_3 + x_6 + x_9 + x_{12} \leq 1]$	$[x_2 + x_5 + x_8 + x_{11} \leq 1]$	$[x_3 + x_6 + x_9 + x_{12} \leq 1]$
	$[x_3 + x_6 + x_9 + x_{12} \leq 1]$	

Now, we will examine two possible sequences of reformulations. In sequence 1, Algorithm MSAT first identifies the core involving these soft clauses: $(x_1 \land x_2 \land x_3)$. It applies cover optimization, and in polynomial time (quadratic) on the size of the core, it can replace (x_1, x_2, x_3) with $(_{(2)}\ \square)$ and $(x_1 \lor x_2 \lor x_3)$. This also corresponds to applying the *At Most One MaxSAT* rule for preprocessing in MaxSAT. The solver repeats the same process on the next cores: $(x_4 \land x_5 \land x_6)$, $(x_7 \land x_8 \land x_9)$, and $(x_{10} \land x_{11} \land x_{13})$. At this point, the current MaxSAT formula

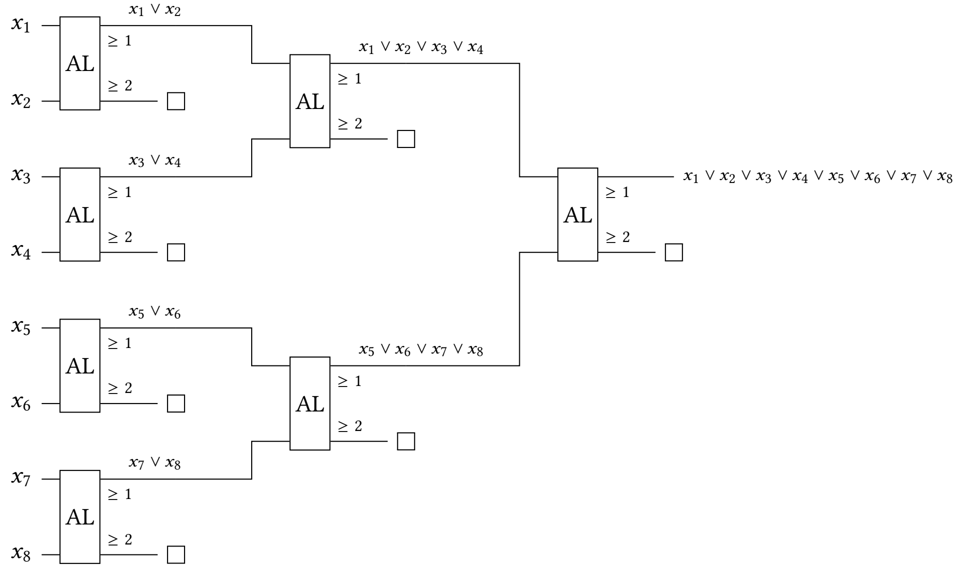


Fig. 2. Example of the comparators chain generated by a Core-guided MaxSAT solver (Algorithm MSAT) on the MaxSAT version of PHP_1^8 (at most one soft constraint can be satisfied).

has a lower bound of 8, below the optimum. Notice that if we carefully inspect the conjunction of soft and hard clauses (as a SAT formula), we can see the encoding of a Pigeon-Hole Principle (PHP) with 3 holes and 4 pigeons.

In sequence 2, following the same reformulation strategy as before, the sequence of cores begins with: $(x_1 \wedge x_4 \wedge x_7 \wedge x_{10})$, which helps us replace soft clauses x_1, x_4, x_7, x_{10} with $(_{(3)} \square)$ and $(x_1 \vee x_4 \vee x_7 \vee x_{10})$. We repeat this for the cores $(x_2 \wedge x_5 \wedge x_8 \wedge x_{11})$ and $(x_3 \wedge x_6 \wedge x_9 \wedge x_{12})$. The resulting MaxSAT instance has a trivial lower bound of 9, and the conjunction of soft and hard clauses becomes a trivially satisfiable formula for the last SAT call that returns True. We just need to satisfy the ith literal from the ith soft clause. Therefore, 9 is the optimum.

If we generalize the previous example to n pigeons (with $n * (n-1)$ original soft clauses), strategy for sequence 1 solves n MaxSAT versions of the PHP_1^{n-1} problem with an accumualted cost of $n * (n-2)$ (in $O(n^3)$ time) and ends up generating a PHP_{n-1}^n formula which is exponential for resolution and therefore for current SAT solvers.

However, strategy in sequence 2 solves $n-1$ MaxSAT versions of the PHP_1^n problem with an accumulated cost of $(n-1)^2$ (in $O(n^3)$ time) and ends up generating a trivial satisfiable SAT formula.

□

7.2 Empirical Evidence on the Hardness of Alternative Core-Sequences

Since the order in which we find the cores is crucial, we run a simple experiment consisting of randomly shuffling the input soft clauses. This will affect the way the assumptions are provided to the SAT solver, generating different cores and, consequently, different sequences. In particular, we use our MaxSAT solver (version MSAT-D4-ALK$_6$-ext2 as described in Section 8) with the default configuration and EvalMaxSAT (Berg, Järvisalo, Martins, Niskanen, and Paxian 2024), see the next section for the experimentation settings.

We run each solver on four soft-orderings of the same input instance: the original order in the input MaxSAT instance and three additional shuffles of just the soft clauses. We present a scatter plot in Figure 3 showing the best and worst execution times for each instance. As we can observe, just by shuffling the order of the input soft

constraints, we can get an important variance. In particular, there are extreme data points where one soft-ordering (out of the 4) timeouts (3600) while another order is solved in less than 100 seconds. Red data points mean some version caused a timeout.

MSAT-D4-ALK$_6$-ext2 (EvalMaxSAT) solves 408 (418) with the original soft-ordering. The Virtual best solver (taking the best result of all four soft-orderings) for MSAT-D4-ALK$_6$-ext2 (EvalMaxSAT) solves 416 (422). There are 35 (10) for MSAT-D4-ALK$_6$-ext2 (EvalMaxSAT) for which at least one soft-ordering out of the 4 timeouts while another one solves the instance.

As a summary, the soft-ordering does affect, but it has more impact on MSAT-D4-ALK$_6$-ext2. We conjecture this has to do with the way assumptions are provided to the SAT solver within the algorithm (see description on how assumptions are added to the SAT solver in Section 6).

7.3 Using Machine Learning (ML) to Trigger Alternative Core-Sequences.

In this work, we explore using alternative configurations of the same solver, as modifying parameters of the underlying algorithm will result in the solver naturally exploring alternative core sequences. Notice that we can apply this strategy at any point in the current sequence of cores that has been reformulated by the Core-guided solver.

In particular, our approach generates a portfolio that uses a ML model to predict the expected runtime of each configuration based on the following features of a MaxSAT instance: the number of clauses, the number of soft and hard clauses, the number of variables, soft and hard clauses statistics (average, standard deviation, minimum, and maximum), variable graph features, balance features (ratio of positive to negative literals per clause, ratio of positive to negative occurrences of each variable), fraction of unary, binary, and ternary clauses, and statistics of horn clauses.

Our training set for the ML model is the set of instances from the unweighted track of the MaxSAT evaluation 2023 (Berg, Järvisalo, Martins, and Niskanen 2023), and our test set is the instances from the same track in the subsequent year.

First, we gather the values of the different features for each instance, and then we run the different configurations of our MaxSAT solver on the training set, which provides us with the information in terms of runtime (CPU time). Then, we build a multi-output model, where the input is a vector with the features' values of the instance, and the output is a vector with the expected runtime for each configuration (scaled using a logarithmic function). From this output vector, we will select the solver with the lowest predicted runtime.

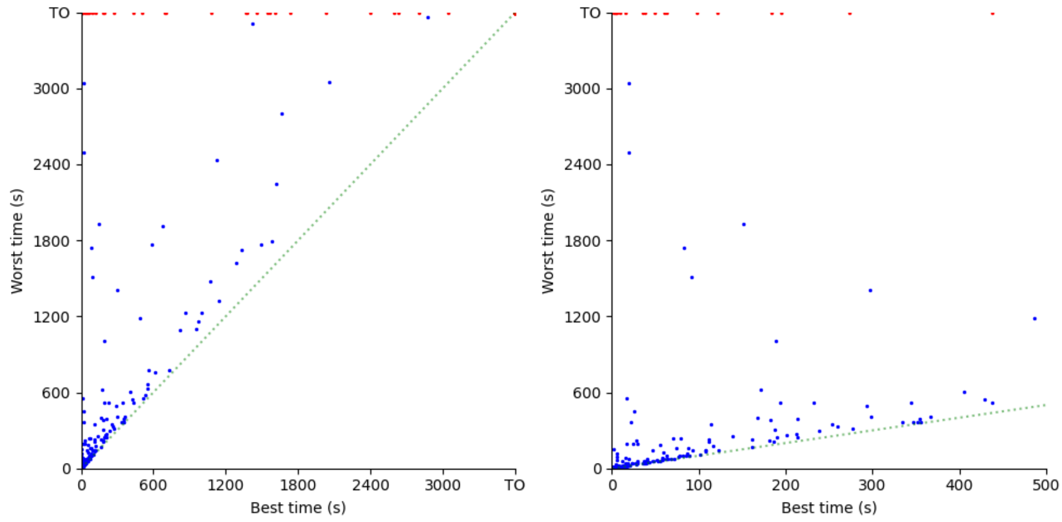
In particular, we train an *ExtraTreesRegressor* model from the *scikit-learn* framework (Pedregosa et al. 2011). First, we use a grid search on the parameters for this model, with a cross-validation procedure to select the best ones. Once we have selected the best parameters on the training set, we train the final model, which we then evaluate on MSE24. Results are reported in the next section.

Notice that we apply this procedure as a preprocessing step, but it can be turned into an inprocessing procedure. For example, we can execute it between iterations of the MaxSAT solver, even augmenting the ML model with features about the sequence of cores reformulated so far. We leave this research avenue as future work.

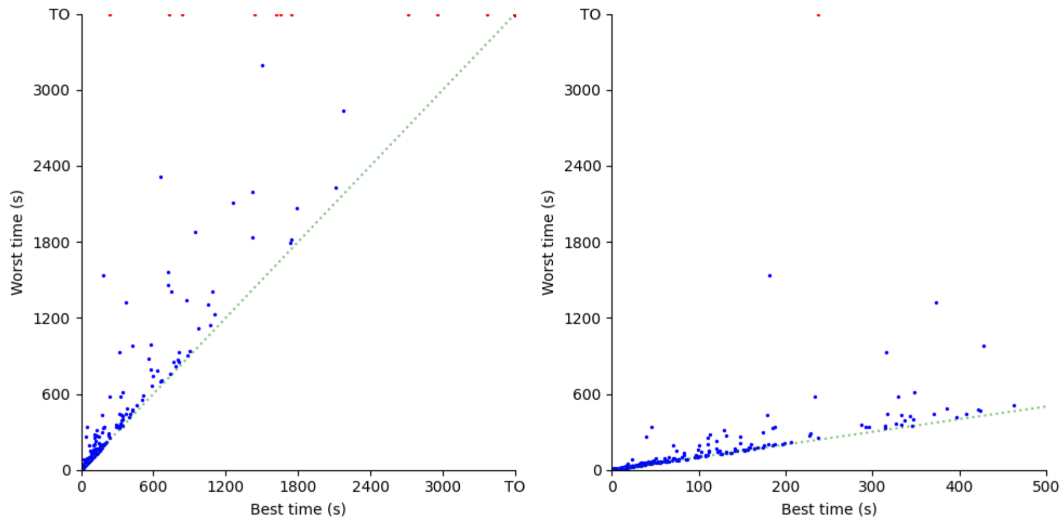
8 Experimental Results

We refer to our solver as MSAT, which is an implementation in Python of the Algorithm MSAT presented in Section 6. The underlying incremental SAT solver is CaDiCaL version 2.0.0 (Biere et al. 2024).

We compare with the best-performing solvers from MSE 2024 in the unweighted category that are not portfolios from different families: WMaxCDCL, MaxCDCL (C. M. Li, Xu, et al. 2021; C.-M. Li et al. 2022) (B&B), and EvalMaxSAT (Berg, Järvisalo, Martins, Niskanen, and Paxian 2024) (Core-guided). For EvalMaxSAT, we



(a) Comparison on MSAT-D4-ALK$_6$-ext2, with a time limit of 1h and zooming in to 500s.



(b) Comparison on EvalMaxSAT, with a time limit of 1h and zooming in to 500s.

Fig. 3. Best versus worst runtime for alternative order of soft constraints. Alternative orders are achieved by shuffling the soft clauses before inputting them to the MaxSAT solver. Red dots represent timeouts.
Notice that dots on the top-left corner represent instances for which a sequence of cores leads to solving the instance in seconds, while another sequence needs more than an hour.

experiment both with and without its own version of AMO preprocessing (Ignatiev, Morgado, et al. 2019), EvalMaxSAT and EvalMaxSAT-noAMO, respectively.

We also compare Core-guided RC2 algorithm (Ignatiev, Morgado, et al. 2019) (RC2A and RC2B) which is implemented in python on top of pysat package (Ignatiev, Tan, et al. 2024), CSGSS (Ihalainen, Berg, et al. 2021) (Python version), CGSS2 (Ihalainen 2022) (C++ version), MaxHS (Davies and Bacchus 2011) (hybrid MIP and SAT), UWr (Piotrów 2020), Eva500a (Narodytska and Bacchus 2014) (Core-guided implementing PMRes), Pacose (Paxian and Becker 2020) (Model-guided MaxSAT solver) and OpenWBO-OLL, OpenWBO-Res (Martins et al. 2014).

The unweighted category at MSE 2024 includes MaxSAT instances where all soft clauses have the same weight and may contain hard clauses. This category contains 553 instances, including industrial and crafted instances from a total of 42 families. One particular family, consisting of 25 crafted instances from *Minimizing Pentagons in the Plane* (Berg, Järvisalo, Martins, Niskanen, and Paxian 2024; Subercaseaux et al. 2024), is solved exclusively by MaxCDCL solvers. This is why MaxCDCL and WMaxCDCL have two outcomes with and without this family.

The executions of the MaxSAT solvers are run on a computational cluster composed of nodes with two AMD 7402 processors (each with 24 cores at 2.8 GHz) and 1T GB of RAM per node, managed by Sun Grid Engine (SGE). Each execution is allocated 3600 seconds of CPU time and 32 GB of memory, as in the exact track from the latest MaxSAT Evaluation (Berg, Järvisalo, Martins, Niskanen, and Paxian 2024).

In Table 1, we consider different variations of our solver that differ in how we build the cardinality constraint used to reformulate the unsatisfiable core. To describe the different versions we use the template MSAT-μ-ALK$_\tau$-σ, where $\mu \in \{D4, D3\}$ is the MaxSAT rule used to create the cardinality constraint, $\tau \in \{1, 6, all\}$ is the number of output comparator for algorithm ALK$_n$, and $\sigma \in \{ext1, ext2, ext3\}$ is the way we create the auxiliar variables as described in Example 5.6. We also consider using the Totalizer encoder from the pysat (Ignatiev, Tan, et al. 2024).

By default, the input soft clauses are fully reified into a new auxiliary variable (as in the *Extension* rule) if they are not unary. Additionally, we use the *freeze* method from the CaDiCaL solver to prevent it from eliminating the auxiliary variables representing soft clauses. Before starting the MaxSAT solving process, we call the *simplify* method of CaDiCaL to simplify the hard clauses. We postpone the reformulation of soft cores (Marques-Silva and Planes 2008) and only reformulate those of minimum size. We have seen that disabling postpone over MSAT-D4-Totalizer-ALK$_6$-ext2 solves 410 (with a slightly worse PAR2 score) instances, and reformulating all the cores solves 400 instances.

Other options available but not enabled by default are: computing the initial upper bound with an external anytime solver, core trimming and core minimization (Ignatiev, Morgado, et al. 2019), preprocessing failed soft clauses, and preprocessing At-Most-One (Ignatiev, Morgado, et al. 2019) and other preprocessing techniques.

Looking at the results, we first notice that it is always better to produce at least 6 comparators for rules *D3* and *D4*, but not all possible comparators. This makes sense since we need to find a balance between the number of comparators (good for propagation) and the size of the resulting encoding. We also notice that the best combination is *D4* with $ext2$ for translating into CNF. The Totalizer option, despite being used in several state-of-the-art solvers, is not as competitive.

Additionally, we try a hybrid approach that applies the Totalizer if the size of the core is less than 1000, and MSAT-D4-ALK$_6$ otherwise, referred to as MSAT-Totalizer-D4-ALK$_6$, which performs slightly better. In summary, even though our solver does not use several well-known strategies in the default configuration, it compares reasonably well with the rest of the state-of-the-art MaxSAT solvers. In fact, we can further explore a strategy where, when reformulating the unsatisfiable core, we order the inputs to the cardinality encoder based on the model corresponding to the best upper bound found so far, in the flavor of (Ansótegui and Gabàs 2017) on MaxSAT phase saving. The first inputs are those input soft constraints that are satisfied in the model. This leads us to solve 413 instances.

Table 1. Number of solved instances at the unweighted track of MSE 2024 in 1h with 32GB.

Solver	MSE24
EvalMaxSAT	418
MaxCDCL	415 (390)
WMaxCDCL	414 (389)
MaxHS	409
UWr	406
EvalMaxSAT-noAMO	404
RC2B	399
CGGS	397
CGGS2	394
RC2A	393
OpenWBO-Res	386
OpenWBO-OLL	363
Eva500a	362
Pacose	356

Solver	MSE24
MSAT-Totalizer-D4-ALK$_6$-ext2-ML	425
MSAT-Totalizer-D4-ALK$_6$-ext2-3configs-4random	418
MSAT-Totalizer-D4-ALK$_6$-ext2-3configs	415
MSAT-Totalizer-D4-ALK$_6$-ext2-4random	414
MSAT-Totalizer-D4-ALK$_6$-ext2	410
MSAT-D4-ALK$_6$-ext2	408
MSAT-D3-ALK$_6$-ext3	403
MSAT-Totalizer	403
MSAT-D4-ALK$_1$-ext2	402
MSAT-D3-ALK$_1$-ext3	397
MSAT-D4-ALK$_{all}$-ext2	395
MSAT-D4-ALK$_6$-ext1	391
MSAT-D4-ALK$_1$-ext1	381

At this point, we claim that our contribution has enough value from an empirical point of view, since with a simple scheme — which can also easily produce certificates — we can already outperform or be competitive with most of the best MaxSAT solvers in the last years. However, we are aware that the reader may wonder whether we can surpass the winner of the last MSE. Notice that these solvers are highly optimized and tuned for the MSE, so this requires a complex combination of many techniques and adjustments. In the next experiments, we show how we can reach this goal and show evidence that there is further room for future improvements.

As discussed in Section 7, we believe it is more important to explore possible alternative sequences of cores rather than optimizing a given one. Notice that, as observed in Figure 3, just by randomly changing the order of the soft clauses in the initial MaxSAT instance, we can observe enough variance in the runtime. In MSAT-Totalizer-D4-ALK$_6$-4random, we execute the MaxSAT solver with the default order and four random orders, limiting the conflicts budget of the SAT solver to 100. After the executions, we select the one with the highest achieved lower bound. In MSAT-Totalizer-D4-ALK$_6$-3configs, we use three alternative configurations of the solver, and in MSAT-Totalizer-D4-ALK$_6$-3configs-4random, we use the three configurations and also the 4 random orders of the soft clauses. We have not fully explored these combinations, which may lead to further improvements.

In MSAT-Totalizer-D4-ALK$_6$-ML, we apply the approach presented in Section 7.3 with 17 alternative configurations. Notice that we are only exploring possible alternatives at the root node of the tree of possible sequences. There is room for improvement in exploring alternative sequences deeper in the tree.

The selected configurations also include additional techniques we explored during the design of our algorithm. What we seek with these additional techniques is just to get more variance, i.e., alternative ways of exploring the tree of core-sequences that we can later exploit with our ML approach.

These additional techniques include:

- External UB (*nuUB*): Computes an initial UB using the anytime solver Nu (Chu et al. 2023)
- True First in Cardinality constraints (*TrueFirst*): When reformulating, the first inputs to the cardinality encoder are the literals satisfied in the model.

- False First in Cardinality constraints (*FalseFirst*): When reformulating, the first inputs to the cardinality encoder are the literals falsified in the model.
- Interleave literals in Cardinality constraints (*Interleave*): When reformulating, the first inputs are interleaved (alternating true and false literals in the model).
- Phase saving (*phaseSaving*): Enables phase saving (Ansótegui and Gabàs 2017).
- Unit Propagation (UP) after applying *D4* (*propagateAndLit*): Once we have applied *D4* and *Extension* rules, we force the solver to perform UP on the literal that represents the conjunction generated by *D4*.
- Community-guided MSAT solving: We take the hard clauses after executing line 2 in Algorithm MSAT. We build a Variable Incidence Graph (VIG) on the hard clauses (Ansótegui, Giráldez-Cru, et al. 2012; Ansótegui and Levy 2011). Then we run VieClus (Biedermann et al. 2018) for 10 seconds on the VIG graph to detect communities of variables. We filter out, from each community, variables not corresponding to literals in structure *softs*. Then, we have two options:
  – (*communityIndependent*) Force Algorithm MaxSAT to solve to optimality each subproblem induced by each community. In essence, this corresponds to applying our Algorithm MaxSAT just to the literals in *softs* corresponding to the variables in a particular community. Finally, just run Algorithm MaxSAT on the joined reformulations of every community, which solves the input MaxSAT problem.
  – (*communityCumulative*) Incrementally add to *softs* the literals corresponding in the next unprocessed community and solve to optimality by applying Algorithm MaxSAT. When the last community is processed, Algorithm MaxSAT solves the input MaxSAT problem.

In particular, selected configurations are:

- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-3configs*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-4random*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2*
- *MSAT-Totalizer*
- *MSAT-D4-ALK*$_1$*-ext2*
- *MSAT-D4-ALK*$_{all}$*-ext2*
- *MSAT-D4-ALK*$_6$*-ext2*
- *MSAT-D4-ALK*$_6$*-ext1*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-TrueFirst*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-FalseFirst*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-Interleave*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-phaseSaving*
- *MSAT-D4-ALK*$_6$*-ext2-phaseSaving*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-communityIndependent*
- *MSAT-Totalizer-D4-ALK*$_6$*-ext2-communityCumulative*
- *MSAT-D4-ALK*$_6$*-ext2-propagateAndLit*
- *MSAT-D4-ALK*$_6$*-TrueFirst-NuUB*

Table 2 compares the performance of the MSAT approaches (first subcolumn) against EvalMaxSAT (second subcolumn).

In columns 1 and 2, we summarise the performance *per family* in the MSE24, which has 42 families. Column *family % avg max* reports the average percentage of solved instances per family considering the maximum number of solved, whereas column *family % avg all* reports the same considering the total number of instances per family. We observe that MSAT-Totalizer-D4-ALK$_6$-ext2-ML and MSAT-Totalizer-D4-ALK$_6$-ext2-3configs-4random outperform EvalMaxSAT in both metrics. We also observed that none of our compared MaxSAT solvers dominate in a particular family, showing the robustness of our approaches.

On the other hand, columns 3 and 4 report the total number of solved instances where there is a difference in running time of a factor of 5 (column *factor 5*) and a factor of 10 (column *factor 10*). We exclude those instances that were solved in less than 5 seconds. All the tested MSAT approaches outperform EvalMaxSAT in these two metrics, even though EvalMaxSAT is developed in C++ and our approaches are in Python.

Regarding the complementarity of solver families, our best Core-guided approach (excluding the ML-based one) solves 36 instances that B&B MaxCDCL can't, while MaxCDCL solves 28 our approach can't. We also solve 26 instances that MaxHS can't, while MaxHS solves 12 that our approach can't.

Table 2. Comparison of MSAT solver variants against EvalMaxSAT

Solver	family % avg max	family % avg all	factor 5	factor 10
MSAT-Totalizer-D4-ALK$_6$-ext2-ML	0.946 / 0.936	0.769 / 0.758	46 / 13	22 / 9
MSAT-Totalizer-D4-ALK$_6$-ext2-3configs-4random	0.943 / 0.938	0.759 / 0.758	43 / 17	17 / 13
MSAT-Totalizer-D4-ALK$_6$-ext2-3configs	0.934 / 0.946	0.753 / 0.758	47 / 16	20 / 12
MSAT-Totalizer-D4-ALK$_6$-ext2-4random	0.936 / 0.946	0.749 / 0.758	41 / 13	21 / 10
MSAT-Totalizer-D4-ALK$_6$-ext2	0.917 / 0.948	0.745 / 0.758	47 / 11	21 / 10
MSAT-Totalizer	0.901 / 0.948	0.732 / 0.758	45 / 11	21 / 10
MSAT-D4-ALK$_{all}$-ext2	0.871 / 0.948	0.719 / 0.758	40 / 14	13 / 10
MSAT-D4-ALK$_6$-ext2	0.902 / 0.950	0.742 / 0.758	46 / 14	19 / 9
MSAT-D4-ALK$_1$-ext2	0.887 / 0.952	0.728 / 0.758	41 / 13	15 / 9
MSAT-D3-ALK$_6$-ext3	0.894 / 0.952	0.732 / 0.758	49 / 12	20 / 11
MSAT-D3-ALK$_1$-ext3	0.903 / 0.948	0.738 / 0.758	40 / 11	20 / 8
MSAT-D4-ALK$_6$-ext1	0.864 / 0.950	0.711 / 0.758	43 / 11	16 / 9

9 Conclusions and Future Work

In this paper, we shed light on the connection between Core-guided solvers and MaxSAT rules, opening up research avenues for other applications of MaxSAT rules, not only those guided by the detection of unsatisfiable cores. We also indirectly provide the basis for a simple way to certify Core-guided solvers, as a straightforward proof can be written as a sequence including MaxSAT, *Extension*, and *Fold* rules, which we leave for future work.

Finally, we demonstrate that it is crucial to explore alternative sequences of reformulation, aiming to avoid exponentially harder sequences. In this sense, we present a simple yet effective approach based on machine learning techniques. As future work, we plan to explore alternative ML approaches to guide a meta-procedure on the search space of core-sequences.

10 Acknowledgements

This work was supported by the project PROOFS BEYOND (grant number PID2022-138506NB-C21) funded by the AEI.

References

B. Andres, B. Kaufmann, O. Matheis, and T. Schaub. 2012. "Unsatisfiability-based optimization in clasp." In: *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary* (LIPIcs). Ed. by A. Dovier and V. S. Costa. Vol. 17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 211–221.

C. Ansótegui. 2021. *SAT, Gadgets, and MaxSAT*. Invited talk at SAT conference 2021, Barcelona, Spain. (2021).

C. Ansótegui, M. L. Bonet, and J. Levy. Mar. 2013. "SAT-based MaxSAT algorithms." *Artif. Intell.*, 196, (Mar. 2013), 77–105. doi:10.1016/j.artint.2013.01.002.

C. Ansótegui and J. Gabàs. 2017. "WPM3: An (in)complete algorithm for weighted partial MaxSAT." *Artif. Intell.*, 250, 37–57.

C. Ansótegui, J. Gabàs, and J. Levy. Feb. 2016. "Exploiting subproblem optimization in SAT-based MaxSAT algorithms." *Journal of Heuristics*, 22, (Feb. 2016). doi:10.1007/s10732-015-9300-7.

C. Ansótegui, J. Giráldez-Cru, and J. Levy. 2012. "The community structure of SAT formulas." In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 410–423.

C. Ansótegui and J. Levy. 2011. "On the modularity of industrial SAT instances." In: *Artificial Intelligence Research and Development*. IOS Press, 11–20.

R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. 2009. "Cardinality Networks and Their Applications." In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings* (Lecture Notes in Computer Science). Ed. by O. Kullmann. Vol. 5584. Springer, 167–180.

O. Bailleux and Y. Boufkhad. 2003. "Efficient CNF Encoding of Boolean Cardinality Constraints." In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings* (Lecture Notes in Computer Science). Ed. by F. Rossi. Vol. 2833. Springer, 108–122.

K. E. Batcher. 1968. "Sorting networks and their applications." In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 307–314.

J. Berg, M. Järvisalo, R. Martins, and A. Niskanen. 2023. "MaxSAT Evaluation 2023: Solver and Benchmark Descriptions."

J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian. 2024. "MaxSAT Evaluation 2024 : Solver and Benchmark Descriptions." eng. *Department of Computer Science Series of Publications B*, B-2024-2. http://hdl.handle.net/10138/584878.

S. Biedermann, M. Henzinger, C. Schulz, and B. Schuster. 2018. "Memetic Graph Clustering." In: *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA'18)* (LIPIcs). Technical Report, arXiv:1802.07034. Dagstuhl.

A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. 2024. "CaDiCaL 2.0." In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I* (Lecture Notes in Computer Science). Ed. by A. Gurfinkel and V. Ganesh. Vol. 14681. Springer, 133–152. doi:10.1007/978-3-031-65627-9_7.

I. Bonacina, M. L. Bonet, and J. Levy. 2024. "Polynomial calculus for optimization." *Artif. Intell.*, 337, 104208. doi:10.1016/J.ARTINT.2024.104208.

I. Bonacina, J. Levy, and I. M. Liberal. 2025. *Beyond Core-Guided MaxSAT*. https://satcpdp25.github.io/submissions/satcpdp-25_paper_12.pdf. [Accessed 04-11-2025]. (2025).

M. L. Bonet, J. Levy, and F. Manyà. 2007. "Resolution for Max-SAT." *Artif. Intell.*, 171, 8-9, 606–618.

Y. Chu, S. Cai, and C. Luo. 2023. "Nuwls: Improving local search for (weighted) partial maxsat by new weighting techniques." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 4. Vol. 37, 3915–3923.

S. A. Cook and R. A. Reckhow. 1979. "The Relative Efficiency of Propositional Proof Systems." *J. Symb. Log.*, 44, 1, 36–50. doi:10.2307/2273702.

J. Davies and F. Bacchus. 2011. "Solving MAXSAT by Solving a Sequence of Simpler SAT Instances." In: *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings* (Lecture Notes in Computer Science). Ed. by J. H. Lee. Vol. 6876. Springer, 225–239.

M. Davis and H. Putnam. 1960. "A computing procedure for quantification theory." *Journal of the ACM*, 7, 3, 201–215.

K. Fazekas, F. Bacchus, and A. Biere. 2018. "Implicit Hitting Set Algorithms for Maximum Satisfiability Modulo Theories." In: *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings* (Lecture Notes in Computer Science). Ed. by D. Galmiche, S. Schulz, and R. Sebastiani. Vol. 10900. Springer, 134–151. doi:10.1007/978-3-319-94205-6_10.

Y. Filmus, M. Mahajan, G. Sood, and M. Vinyals. 2020. "MaxSAT Resolution and Subcube Sums." In: *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings* (Lecture Notes in Computer Science). Ed. by L. Pulina and M. Seidl. Vol. 12178. Springer, 295–311. doi:10.1007/978-3-030-51825-7_21.

F. Heras and J. Larrosa. 2006. "New Inference Rules for Efficient Max-SAT Solving." In: *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA.* AAAI Press, 68–73. http://www.aaai.org/Library/AAAI/2006/aaai06-011.php.

F. Heras, J. Larrosa, and A. Oliveras. 2007. "MiniMaxSat: A New Weighted Max-SAT Solver." In: *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings* (Lecture Notes in Computer Science). Ed. by J. Marques-Silva and K. A. Sakallah. Vol. 4501. Springer, 41–55. doi:10.1007/978-3-540-72788-0_8.

A. Ignatiev, A. Morgado, and J. Marques-Silva. Sept. 2019. "RC2: an Efficient MaxSAT Solver." English. *Journal on Satisfiability, Boolean Modeling and Computation*, 11, 1, (Sept. 2019), 53–64. doi:10.3233/SAT190116.

A. Ignatiev, Z. L. Tan, and C. Karamanos. 2024. "Towards Universally Accessible SAT Technology." In: *SAT*, 4:1–4:11. doi:10.4230/LIPICS.SAT.2024.16.

H. Ihalainen. 2022. "Refined core relaxations for core-guided maximum satisfiability algorithms." Ph.D. Dissertation. Master's thesis, University of Helsinki. http://hdl. handle. net/10138/351207.

H. Ihalainen, J. Berg, and M. Järvisalo. 2021. "Refined Core Relaxation for Core-Guided MaxSAT Solving." In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by L. D. Michel. Vol. 210. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:19. ISBN: 978-3-95977-211-2. doi:10.4230/LIPIcs.CP.2021.28.

H. Ihalainen, J. Berg, and M. Järvisalo. 2024. "Unifying SAT-Based Approaches to Maximum Satisfiability Solving." *J. Artif. Intell. Res.*, 80, 931–976. doi:10.1613/JAIR.1.15986.

A. Kügel. 2010. "Improved Exact Solver for the Weighted MAX-SAT Problem." In: *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010* (EPiC Series in Computing). Ed. by D. L. Berre. Vol. 8. EasyChair, 15–27. doi:10.29007/38LM.

J. Larrosa and F. Heras. 2005. "Resolution in Max-SAT and its relation to local consistency in weighted CSPs." In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005.* Ed. by L. P. Kaelbling and A. Saffiotti. Professional Book Center, 193–198.

C. M. Li, F. Manyà, N. O. Mohamedou, and J. Planes. 2010. "Resolution-based lower bounds in MaxSAT." *Constraints An Int. J.*, 15, 4, 456–484. doi:10.1007/S10601-010-9097-9.

C. M. Li, F. Manyà, and J. Planes. 2011. "New Inference Rules for Max-SAT." *CoRR*, abs/1111.0040. http://arxiv.org/abs/1111.0040 arXiv: 1111.0040.

C. M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He. 2021. "Combining clause learning and branch and bound for MaxSAT."

C.-M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He. 2022. "Boosting branch-and-bound MaxSAT solvers with clause learning." *AI Communications*, 35, 2, 131–151.

J. Marques-Silva and J. Planes. Apr. 2008. "Algorithms for Maximum Satisfiability Using Unsatisfiable Cores." In: (Apr. 2008), 408–413. ISBN: 978-3-9810801-4-8. doi:10.1109/DATE.2008.4484715.

R. Martins, V. Manquinho, and I. Lynce. 2014. "Open-WBO: A Modular MaxSAT Solver," in: *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* (Lecture Notes in Computer Science). Ed. by C. Sinz and U. Egly. Vol. 8561. Springer, 438–445.

A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. 2013. "Iterative and core-guided MaxSAT solving: A survey and assessment." *Constraints*, 18, 478–534.

N. Narodytska and F. Bacchus. 2014. "Maximum Satisfiability Using Core-Guided MaxSAT Resolution." In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.* Ed. by C. E. Brodley and P. Stone. AAAI Press, 2717–2723.

T. Paxian and B. Becker. 2020. "Pacose: An iterative SAT-based MaxSAT solver." *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, 12.

F. Pedregosa et al.. 2011. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research*, 12, 2825–2830.

M. Piotrów. 2020. "UWrMaxSat: Efficient Solver for MaxSAT and Pseudo-Boolean Problems." In: *32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020, Baltimore, MD, USA, November 9-11, 2020.* IEEE, 132–136.

C. Sinz. 2005. "Towards an Optimal CNF Encoding of Boolean Cardinality Constraints." In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings* (Lecture Notes in Computer Science). Ed. by P. van Beek. Vol. 3709. Springer, 827–831.

B. Subercaseaux, J. Mackey, M. J. H. Heule, and R. Martins. 2024. "Automated Mathematical Discovery and Verification: Minimizing Pentagons in the Plane." In: *Intelligent Computer Mathematics: 17th International Conference, CICM 2024, Montréal, QC, Canada, August 5–9, 2024, Proceedings.* Springer-Verlag, Montreal, QC, Canada, 21–41. ISBN: 978-3-031-66996-5. doi:10.1007/978-3-031-66997-2_2.

G. S. Tseitin. 1983. "On the Complexity of Derivation in Propositional Calculus." In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970.* Ed. by J. H. Siekmann and G. Wrightson. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. ISBN: 978-3-642-81955-1. doi:10.1007/978-3-642-81955-1_28.

Received 13 June 2025; accepted 16 November 2025