Analyzing Reformulation Performance in Core-Guided MaxSAT Solving

André Schidler
University of Freiburg, Germany
TU Wien, Austria

Stefan Szeider
TU Wien, Austria

Abstract

Core-guided algorithms like OLL are among the best methods for solving the Maximum Satisfiability problem (MaxSAT). Although some performance characteristics of OLL have been studied, a comprehensive experimental analysis of its reformulation behavior is still missing. In this paper, we present a large-scale study on how different reformulations of a MaxSAT instance produced by OLL affect solver performance. By representing these reformulations as a directed acyclic graph (DAG), we isolate the impact of structural features – such as the size and interconnectivity of unsatisfiable cores – on solver runtime. Our extensive experimental evaluation of over 600k solver runs reveals clear correlations between DAG properties and performance outcomes. These results suggest a new avenue for designing heuristics that steer the solver toward more tractable reformulations. All OLL DAGs and performance data from our experiments are publicly available to foster further research.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases maximum satisfiability, OLL, core-guided

Digital Object Identifier 10.4230/LIPIcs.SAT.2025.26

Supplementary Material *Dataset*: `https://doi.org/10.5281/zenodo.15584371`

Funding This work was supported by Austrian Science Fund (FWF) grants 10.55776/P36420 and 10.55776/COE12, as well as an Amazon Research Award (Fall/2023).

1 Introduction

MaxSAT (Maximum Propositional Satisfiability) extends SAT by optimizing solutions under constraints. Applications span software analysis [26], post-silicon fault detection [28], concurrent program debugging [16], smartphone app security [13], and computer-aided design [23, 24]. As the complexity of these domains grows, the need for efficient and scalable MaxSAT solvers becomes increasingly important.

Core-guided algorithms are among the most effective approaches for solving MaxSAT instances, which iteratively refine the problem using information extracted from unsatisfiable cores. One prominent example of such an algorithm is *OLL* [1, 22] which has consistently performed well in MaxSAT competitions [9][1]. OLL's ability to compute optimal solutions with proof logging [7] makes it both efficient and verifiable.

Despite its strengths, OLL's performance can vary significantly depending on internal design choices, particularly the reformulations it generates during solving. Reformulations play a pivotal role in the algorithm's efficiency by converting the MaxSAT instance into a series of SAT instances. However, not all reformulations are equal: some result in SAT instances that are computationally hard to solve, leading to longer runtimes or solver failures.

[1] `https://maxsat-evaluations.github.io/`

Several known factors affect the runtime of OLL: (i) instance-specific reasons, such as the SAT instance being inherently hard to solve; (ii) heuristic-specific reasons, where the heuristics used during OLL take too much time; (iii) reformulation-specific reasons, where the SAT instance becomes too hard due to the reformulation; and (iv) SAT solver variability, which may also impact performance, as different solvers behave differently.

This paper focuses on reason (iii), examining how reformulations alone can lead to significant variations in runtime. By isolating this factor, we aim to understand why some reformulations perform better than others, enabling targeted improvements in OLL's implementation.

To achieve this, we propose a framework for analyzing reformulations using the *OLL DAG*, a directed acyclic graph representation of the reformulations. The OLL DAG abstracts away heuristic-specific and instance-specific issues, allowing us to focus exclusively on the structural properties of the reformulations and their impact on solver runtime. Our results on the correlation between OLL DAG properties and solver performance pave the way for analyzing other OLL-specific features, such as heuristics, by observing their influence on the OLL DAG structure.

Our main contributions are as follows:

1. A framework for analyzing the impact of different OLL reformulations on solver performance using a graph-based representation.
2. A systematic, large-scale experimental analysis based on the structure of OLL DAGs, incorporating over 600 000 abstract OLL runs with runtimes of up to five hours.
3. A publicly available collection of over 100 000 OLL DAGs and their associated performance data intended to foster further research into MaxSAT solving.
4. Insights into how heuristics can be evaluated and improved by analyzing their impact on OLL DAG structure.

By isolating the complexity introduced by reformulations, we provide actionable insights into improving solver performance and developing new heuristics that steer OLL toward simpler, more efficient reformulations.

1.1 Related Work

Solvers, and by extension MaxSAT algorithms, are evaluated each year at the annual MaxSAT evaluation [10]. In the course of solver development, much empirical work is done by the developers improving their solvers. Unfortunately, this work is usually focused on a single solver and not published. Little empirical published research exists, apart from an analysis of using different SAT solvers in the core-guided solver RC2 [17].

The impact of reformulations has been evaluated on a theoretical level. Bacchus and Narodytska [5] provided an early formalization of how cores discovered by core-guided algorithms relate to the original formula, showing that each core found corresponds to a union of cores from the input formula. They identified residue subsumption as a potential optimization where some solutions of the cardinality constraints could be ignored while maintaining correctness. Building on this work, Narodytska and Bjørner [25] conducted a comprehensive analysis of both cores and correction sets, establishing key properties like MinHS-monotonicity that connect core-guided and MinHS-style approaches. They proved that cores extracted by core-guided solvers possess the same important characteristics as those found by implicit hitting set methods, demonstrating an intrinsic connection between these different solving paradigms. Together, these papers established much of the theoretical framework for understanding modern core-guided MaxSAT solving available so far. However, these papers do not contain any empirical analysis and are, therefore, orthogonal to the work in this paper.

2 Preliminaries

Propositional Satisfiability

A literal is a Boolean variable v or its negation $\neg v$, a clause C is a disjunction of literals $C = \ell_1 \vee \cdots \vee \ell_k$, and a formula F in conjunctive normal form (CNF) is a conjunction of clauses $F = C_1 \wedge \cdots \wedge C_m$. We view clauses as a set of literals and a formula as a set of clauses. A variable assignment σ for the variables in F maps each variable to either 1 or 0 corresponding to *true* and *false*. We extend assignments beyond variables as follows. The value assigned to a negated variable is $\sigma(\neg v) = 1 - \sigma(v)$, the value of a clause C is $\sigma(C) = \max_{\ell \in C} \sigma(\ell)$, and the value of a formula is $\sigma(F) = \min_{C \in F} \sigma(C)$. A *model* is an assignment σ such that $\sigma(F) = 1$ and a formula is *satisfiable* if it has a model, and, otherwise, *unsatisfiable*.

MaxSAT

A Maximum Satisfiability (MaxSAT) instance (F, O) consists of a CNF formula F, that we assume is satisfiable, and an objective function of its variables $O = \mathit{Var} \to \mathbb{N}_0$, where Var refers to F's variables. Whenever $O(v) > 0$ for a variable v, we say v is an objective variable. We use b to denote objective variables and v for general variables. An assignment has cost $\Sigma_{v \in \mathit{Var}} O(v) \cdot \sigma(v)$. Hence, any objective variable with $\sigma(b) = 1$ *incurs cost*. An optimal solution to a MaxSAT instance is a model of F with minimum cost. The above definition defines *Weighted Partial MaxSAT* and is equivalent to other definitions, such as using soft and hard clauses. Whenever $O(v) \in \{0, 1\}$ for all $v \in \mathit{Var}$, we say that the instance is *unweighted*.

Cores

Given a MaxSAT instance (F, O), an *unsatisfiable core* C or just *core* is a clause such that C only contains unnegated objective variables and C is entailed by F. Hence, a core states that at least one of its objective variables has to incur cost.

Assumptions

Generally, a SAT solver checks if a given formula is satisfiable or not. Modern SAT solvers also allow for solving under *assumptions* [12, 27]: given a set of literals $\mathcal{A}$ – the assumptions – the SAT solver checks if F has a model σ such that for all $\ell \in \mathcal{A}$ holds that $\sigma(\ell) = 1$, i.e., $\mathcal{A}$ can be extended to a model. Whenever no such model exists, the SAT solver returns a core that is a subset of $\mathcal{A}$.

Limited SAT solving

Limited SAT solving refers to a heuristic use of SAT solvers, where the SAT solver stops after a set limit. This limit can be on the number of conflicts, runtime, or enforced from outside the solver and whenever the limit is reached, the solver returns *unknown*. Limited SAT solving allows controlling how many computing resources are invested into a single SAT call at the price but limited SAT calls might not produce usable results. In this paper we only consider limits on the number of conflicts.

3 The OLL Algorithm

We give a quick overview over the OLL algorithm [1, 22]. The basic OLL algorithm is shown in Algorithm 1. In each iteration, one core is extracted (Line 4) and the instance is reformulated based on this core (Line 7). Each core increases the lower bound by the minimum cost over the core's objective variables (Line 6). The optimal cost has been found as soon as no more cores can be found for the reformulated instance (Line 5).

Algorithm 1 The OLL algorithm for an instance (F, O).

```
1: lb ← 0
2: (F', O') ← (F, O)
3: while True do
4:     C ← extract_core(F', O')
5:     if C = None then return lb end if
6:     lb ← lb + min{ O'(b) : b ∈ C }
7:     (F', O') ← reformulate(F', O', C)
8: end while
```

Next, we describe the two core details of the algorithm: *core extraction* and *reformulation*.

3.1 Reformulation

Reformulation is a central concept of the OLL algorithm. Given an extracted core C, we determine its cost $c = \min\{ O'(b) : b \in C \}$. Since at least one of the objective variables in C has to incur cost, c gives us the minimum cost increase implied by C.

We now compute the reformulated formula F'. Let $n = |C|$. We reformulate F' by adding for each $1 < i \leq n$ a variable b_i^C and clauses that express $\Sigma_{b \in C} b \geq i \rightarrow b_i^C$. Hence, the variable b_i^C is true whenever at least i many objective variables in the core incur cost. The clauses are created by using a cardinality constraint over the core. In this paper, we only consider the *Totalizer* constraints, a common choice in MaxSAT solvers [6, 21].

We adapt O' by setting for each $b \in C$ its cost $O'(b) \coloneqq O'(b) - c$. As this will reduce at least one variable's cost to zero, some variables become non-objectives after reformulation. Further, for each new variable b_i^C we set $O'(b_i^C) \coloneqq c$ expressing that a cardinality increase of C incurs at least cost c. In our implementation, the variables b_i^C with $i > 2$ and the respective constraints are added lazily: they are introduced only when b_{i-1}^C becomes a non-objective, as they are not required before.

3.2 Core Extraction

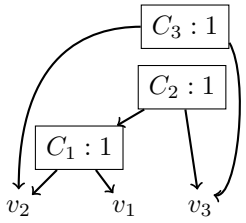
In this paper, we assume that cores are extracted using a SAT solver, which is common in state-of-the-art OLL solvers [9]. Assumptions are used during core extraction: the OLL algorithm assumes that no objective variable in O' incurs cost. The SAT solver then either returns satisfiable or unsatisfiable. In case the result is satisfiable, the model found by the SAT solver is an optimal solution to the reformulated MaxSAT instance that translates to an optimal solution to the original instance and lb – the sum over all core costs – is equal to the optimal cost. Given infinite time, the SAT solver will always return satisfiable whenever lb reaches the optimal value.

In case of an unsatisfiable result, we obtain a core C using the SAT solver and perform the reformulation. Ideally, we want this core to be subset minimal (an *MUS*), but SAT solvers rarely return subset minimal cores. This necessitates an important heuristic in core-guided MaxSAT solving: *core minimization*. We will discuss core minimization and other heuristics commonly used in OLL MaxSAT solving later. First, we introduce the OLL DAG.

4 The OLL DAG

We propose a representation of the OLL reformulations based on a directed acyclic graph (DAG). For a given instance (F, O) and reformulation (F', O'), we define the corresponding OLL DAG $D = (V, E)$ as follows. Let $\mathcal{C}$ be the set of all extracted cores used in (F', O') and $\mathcal{B} = \{ b \in Var : O(b) > 0 \}$. Then, $V = \mathcal{C} \cup \mathcal{B}$ and E contains two types of arcs. The first type consists of arcs $(C, b) \in \mathcal{C} \times \mathcal{B}$, where $(C, b) \in E$ if and only if $b \in C$. Further, the second type of arc is of type $(C, C') \in \mathcal{C} \times \mathcal{C}$ where $(C, C') \in E$ if and only if some $b_j^{C'} \in C$ for $1 < j \leq |C'|$. Hence, the OLL DAG connects cores with the objective variables it contains, or the respective core in case the objective variable was introduced by a reformulation. We annotate D with some extra information: the cost of the core and the number of the core (since cores are extracted sequentially, we give them a consecutive number). Nodes in $\mathcal{B}$ that have in-degree 0 are ignored in our analysis.

▶ **Example 1.** We give an example OLL run and the corresponding OLL DAG. Let $F \coloneqq \{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}$ and $O \coloneqq v_1 + 2 \cdot v_2 + 3 \cdot v_3$. Any model has to set at least two variables to true and the minimum cost would be 3. In the first iteration, we find core $C_1 = \{v_1, v_2\}$, which causes the reformulation $F' \coloneqq \{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}, \{\neg v_1, \neg v_2, b_2^{C_1}\}$, which adds the cardinality semantics for $b_2^{C_1}$. Further, $lb \coloneqq 1$ and $O' \coloneqq v_2 + 3 \cdot v_3 + b_2^{C_1}$: the adaption of the coefficients reduces the coefficient of v_2 and makes v_1 a non-objective. Let the OLL solver find as the second core $C_2 = \{v_3, b_2^{C_1}\}$, for brevity we skip the reformulation as it is analogous to the previous one. This sets $lb \coloneqq 2$. Finally, the OLL solver finds core $C_3 = \{v_2, v_3\}$, which sets $lb \coloneqq 3$. In the next iteration, the OLL solver will assume $\{\neg v_3, \neg b_2^{C_2}, \neg b_2^{C_3}\}$, which can be extended to the model $\{v_1, v_2, \neg v_3, b_2^{C_1}, \neg b_2^{C_2}, \neg b_2^{C_3}\}$. The lb matches the optimal cost of 3 and the corresponding OLL DAG is:



The graph representation allows us to use common graph metrics and analyze if they correlate with successful or unsuccessful solver runs. We analyze several structural properties of D and in this paper, we focus on the following ones, where we consider, where possible, the maximum, minimum, average, and standard deviation over all nodes in the graph. We refer to these as *metrics*.

- The number of cores $|\mathcal{C}|$.
- Let $D[\mathcal{C}]$ be the subgraph induced by $\mathcal{C}$. We define the *width* of D as the size of the largest antichain of the partial order defined by $D[\mathcal{C}]$.
- The in-degrees of the nodes in D.

- The number of and length of source-sink paths. The *depth* is the maximum length over all source-sink paths.
- The number of weakly connected components.
- The out-degrees of the nodes in D, which corresponds to the core sizes. We disregard nodes with out-degree 0 from aggregates.
- The cost of the cores and the objective variables in the cores.
- The *number of combinations*: for a core C, let i be the smallest i such that $O'(b_i^C) > 0$, then the number of combinations for C is $\binom{|C|}{i-1}$. This is the number of assignments to variables in C that satisfy $\Sigma_{b \in C} < i$.
- We define the *child-degree* of a node as the sum over the in-degrees of its children.
- How many nodes can reach a specific node. Let $r(v)$ be the number of nodes that reach $v \in V$, then we define the *interconnectedness* of D as $\frac{\Sigma_{v \in V} r(v)}{|V| \cdot (|V|-1)}$.

Limitations

The OLL DAG is an abstraction that does not capture two important sources that can cause long runtimes in MaxSAT solving: (i) the runtime of the heuristics is not captured and particularly core minimization can take the majority of the runtime, and (ii) the OLL DAG does not contain instance specific information, although different instances will result in different OLL DAGs. Hence, the OLL DAG is only a good model for analyzing complexity that arises from the reformulations. In our view, it is important to analyze this aspect, and we think the OLL DAG is a good option as it abstracts other sources of complexity away.

4.1 OLL Heuristics

In this section, we discuss several common heuristics used to speed up OLL MaxSAT solving and how they are expected to impact the OLL DAG.

Core Minimization [19]

Core minimization aims to convert a core into an MUS using SAT solving under assumptions. Let C be the core to be minimized, then, core minimization tests for each $b \in C$ whether $C' = C \setminus \{b\}$ is a core. In case the SAT solver returns unsatisfiable, C' is indeed a core and core minimization sets $C \coloneqq C'$, otherwise, the process is continued with an unchanged C. Usually the SAT calls are limited and core minimization may not result in an MUS. We expect that core minimization impacts most metrics, as the core size impacts the nodes' degrees and which nodes are connected, the number of combinations, and the cost of the cores.

Shuffling

Shuffling is a simple method that can also reduce the size of cores before core minimization. The order in which the assumptions are given to the SAT solver can change the core the SAT solver returns. Hence, after finding a core C, the objective variables in C are shuffled repeatedly and for each reordering the SAT solver extracts a new core. This can lead to the SAT solver extracting a new core $C' \subset C$. The expected impact on the OLL DAG is the same as for core minimization.

Disjoint Cores / Weight Aware Core Extraction [8]

Instead of reformulating the instance immediately after extracting a core, we only adjust the costs of the objective variables occurring in the core, without changing F' or introducing new variables. We repeat core extraction with limited SAT calls until no more cores can be extracted. Afterwards, we finish reformulating the instance for all remaining cores at once. Disjoint Cores is expected to increase the width of D, as extracting k cores before reformulation guarantees a width of at least k. Furthermore, this should decrease the depth of D.

Core Exhaustion [3]

A core states that at least one objective variable in the core has to incur cost. Core Exhaustion tries to establish if more than one objective variable has to incur cost. After reformulating the instance based on core C, starting with $i = 2$ we perform a limited SAT call assuming that b_i^C does not incur cost. Whenever the SAT solver returns unsatisfiable, $\{b_i^C\}$ is a core, we increment i and repeat the process until the SAT solver does not return unsatisfiable or $i > |C|$. Core exhaustions are represented as singleton cores in D. We do not count these singleton cores in the number of cores metric.

At-Most-One [15]

At-Most-One assumes for each $b \in \mathcal{B}$ that b does not incur cost and performs unit propagation using only this assumption. Whenever the SAT solver returns unsatisfiable, $\{b\}$ is a core. Otherwise, each objective b' that has value 1 after unit propagation corresponds to a core $\{b, b'\}$. Further, the cores $\{b, b'\}$ can be viewed as edges of an auxiliary graph G, and every clique C of G is also a core that can be exhausted up to $|C| - 1$. At-Most-One repeatedly extracts cliques from G, removing used objective variables from G, and adding C as a core. We expect At-Most-One to create high in-degree nodes.

Stratification [2, 20]

Let $\mathcal{B}'$ be the objective variables defined by O', and *score* $m : \mathcal{B}' \to \mathbb{R}$. Stratification partitions $\mathcal{B}'$ into k classes $(P_1, \dots, P_k)$ based on their score. Core extraction is then split into k phases, where in phase $1 \leq j \leq k$ only the objective variables in $\bigcup_{1 \leq i \leq j} P_i$ are assumed not to incur cost.

A common choice for m is the objective variables' costs O'. The expectation is that this leads to cores introducing new objective variables with high (or low) m, where different choices for m may impact the OLL DAG differently.

Hardening [2]

Whenever we have an upper bound ub on the cost, which is often found during Stratification, we can immediately set certain objective variables to 0. Let $gap = ub - lb$, where lb is the lower bound established by the cores as in Algorithm 1. We can now mark each $(b, c) \in O'$ where $c > gap$ as a non-objective, i.e., $O'(b) \coloneqq 0$, as b incurring cost would exceed the upper bound. Hardening does not impact the OLL DAG, as it does not change the cores being found.

5 Experimental Setup

We use an experimental setup that tries to analyze the impact of the OLL DAG structure independently of the solver that created it, as we are not interested in improving a specific solver but the correlation between OLL DAG structure and successful and fast OLL runs.

We base our experiments on the MaxSAT Evaluation 2023 instance set[2] providing 572 unweighted and 558 weighted instances, as at the time we started our experiments, this was the most recent one available. Although a larger set of instances would be preferable, the comprehensiveness of our experiments requires that we focus on a smaller instance set. Eventually, we gathered over 100 000 OLL DAGs and over 600 000 runtime samples over these OLL DAGs.

We split our experiments into three phases:

1. *Pre-Evaluation* removes instances that are too hard to solve, which reduces the runtime of our experiments.
2. *Sampling* runs the OLL solver using many different configurations and stores the OLL DAGs of these runs.
3. *Timing* measures the runtime of an abstract OLL run which is MaxSAT solver and heuristic independent.

We use our own OLL implementation with all the features described in Section 3. Most importantly, using our implementation, we can vary how all heuristics are applied using parameters. Each run in each phase is limited to five hours. That ensures that each run terminates, but the high timeout also enables slower configurations to complete their run.

5.1 Configurations

We cannot force a solver to produce a specific structure. Hence, we use different configurations aimed towards producing a diverse set of OLL DAGs for our analysis. As the goal is diversity, not all configurations are good configurations in the sense that they will solve instances fast. We vary the limits for the heuristics discussed in Section 4.1 and use several different functions for stratification, including a function that returns a random value.

Concretely, we vary the following parameters.

- The conflict limit for the disjoint core phase, where possible values are for unweighted $\{0, 25\,000, 83\,000, 10\,000\,000\}$ and for weighted $\{0, 2\,500\}$.
- The number of shuffles during core shuffling, possible values are for unweighted $\{0, 5, 100, 250, 500\}$ and for weighted $\{0, 6, 100\}$.
- The conflict limit for core minimization, possible values are for unweighted $\{0, 1, 200, 250, 10\,000\}$ and for weighted $\{0, 1, 150, 2\,000, 10\,000\}$.
- Different scoring functions $\mathcal{B}' \to \mathbb{R}$ where $\mathcal{B}'$ is the set of objective variables in O'. The scoring function is used to determine scores for the order during core minimization, the order in which the assumptions are given to the SAT solver and is used during stratification as described in Section 4.1. Besides the objective variable's coefficient in O' we consider the following scoring functions. We always prioritize $b_i \in O$ over objective variables introduced by reformulation. The remaining objective variables $b_j^C \in O'$ are then ordered based on:
  - cost of C, here, $b_i \in O$ are also ordered by cost;
  - number of possible combinations as defined in Section 4;

[2] https://maxsat-evaluations.github.io/2023/

  - depth of the subgraph rooted at C;
  - length of the longest path from any source to C;
  - number of descendants of C;
  - number of ancestors of C;
  - in-degree of C;
  - size of C;
  - length of the longest source-sink path containing C;
  - the child-degree of C as defined in Section 4;
  - a random value.

  Further, we consider ordering the objective variables in ascending and descending order.
- For unweighted instances, scoring functions using cost are not considered as cost is uniform. Further, for weighted instances, we consider either cost and then another scoring function, only the cost, or only the scoring function.
- The number of minimization rounds, either one or five. In the first round, the order of elimination is based on the scores. In subsequent rounds, the order is random. The best core according to cost, then size, then metric is chosen, where ties are broken arbitrarily.
- The number of classes for stratification, possible values for unweighted instances are no stratification or 10 classes. For weighted instances we consider the values no stratification, score-based stratification with 3 classes, cost-based stratification with 14 classes, and cost- then score-based stratification with 42 classes.
- The conflict limit for stratification, which is used for weight- and score-based stratification. We always use 2 650 for unweighted instances and 2 650, 100 000, or 1 000 000 for weighted instances.
- Randomized, where the scoring function assigns random scores, and the order of literals in the cardinality constraints is randomized.

5.2 Pre-Evaluation

The goal of the Pre-Evaluation phase is to determine which instances are very unlikely to be solved, as these instances do not provide useful data while taking up a large fraction of the total runtime. We first establish two good default configurations, one for unweighted and one for weighted instances, using the tool SMAC [18]. After running these configurations, we manually created 18 further configurations that complement the initial configurations, with the goal that combined, these 20 configurations would solve as many instances as possible.

After running all configurations, we discard all instances not solved by a single run. In total, 434 unweighted and 443 weighted instances remain.

5.3 Sampling

In the sampling phase we try to gather as many different OLL DAGs from OLL runs as possible. We manually create different configurations by varying the settings as stated in Section 5.1. In total, we use 113 different unweighted configurations of which 20 are based on randomization. Further, we use 129 weighted configurations, of which 37 use randomization.

Not every instance can be solved by every configuration. We expected to sample 49 042 OLL DAGs from unweighted instances and sampled 49 041, as not a single core could be found for one instance and configuration. In total 46 528 of the 49 042 runs were successful in finding the optimal solution. For weighted instances we expected 57 147 OLL DAGs and obtained 57 136, of which 53 148 were successful.

5.4 Timing

Algorithm 2 Timing instance (F, O) and set of cores $\mathcal{C}$.

$t \leftarrow 0$
$(F', O') \leftarrow (F, O)$
for $C \in \mathcal{C}$ **do**
    $t \leftarrow t + \text{time\_SAT\_call}(F', O')$
    $(F', O') \leftarrow \text{reformulate}(F', O', C)$
end for
$t \leftarrow t + \text{time\_SAT\_call}(F', O')$

The last phase establishes the solver-independent runtimes. We abstract away the MaxSAT solver and measure runtime based on only the SAT solver and the OLL DAG. We use Algorithm 2: we iterate over all cores and measure for each call how long a SAT call takes on the reformulated instance with a timelimit of five hours for all SAT calls. Only the sum of SAT runtimes is relevant for our measurement. Afterwards, a single SAT call is made that either results in a satisfiable result, whenever the original OLL run found the optimal solution, and unsatisfiable otherwise, if there is at least one more core.

The SAT call simulates the call necessary to establish the existence of another core. This abstracts away the use of most heuristics discussed in Section 4.1, except for the At-Most-One heuristic. Further, we do not perform hardening, as the upper bound is dependent on a heuristic result, and making insights dependent on the quality of the bound.

We use the SAT solvers Cadical 1.9.5 [11] and Glucose 3 [4] from PySAT 1.8.dev13 [14]. Both SAT solvers are commonly used in state-of-the-art OLL solvers [10]. Since there is some variance in SAT solving runtimes, we time each OLL DAG three times per SAT solver and, thereby, obtain 294 240 unweighted runtime samples and 342 810 weighted samples. Each sample has one of the following results:

1. *Successful*: The optimal solution has been found.
2. *Failed*: The lower bound does not reach the optimal solution and the timing exceeds the timeout.
3. *Inconclusive:* The lower bound does not reach the optimal solution, but timing finishes within the timeout. Hence, some heuristics like core minimization caused the timeout, not the reformulations.

In our analysis, we ignore the runtime of the last call in case it returned satisfiable and focus on the SAT solver calls corresponding to extracted cores.

6 Analysis

The analysis poses several challenges that require special consideration. The main challenge is that each instance defines its own distribution. Consequently, we cannot normalize and create one dataset over all instances, but instead analyze per instance and then aggregate the analyses. The other special consideration is that runtime is likely to be impacted by several structural properties. Hence, even though the value for one metric stays the same, the runtime may still vary.

Failed Runs

We cannot directly compare failed and successful runs, as the corresponding OLL DAGs are not comparable, since one of the DAGs is incomplete and the metrics would differ simply due to the different number of nodes. Since the two DAGs are from the same instance, we have samples from the same distribution and can normalize. We use the sample with the lowest runtime as the baseline. First, we divide each metric for each successful sample by the corresponding metric value of the baseline. Hence, whenever the sample has the same value as the baseline, it gets value 1 and values between 0 and 1 indicate values smaller than the baseline, while values above 1 indicate values above the baseline. Then, for each failed run we consider the *failure point* of the run: the number of cores extracted at the time of failure. We then gather the baseline's metrics for the OLL DAG at the failure point and normalize the failed run using the baseline's metric at this point. Using the fastest run gives stability towards outliers.

Runtimes

Runtimes have a large variance, even when considering only one SAT solver. We reduce noisiness in our results and focus on harder instances, by discarding every instance from our analysis where every run finishes within 300 seconds. The analysis results are similar for any limit that is at least one second[3]. The Glucose and Cadical runtimes are rarely the same and correlate very weakly. We use the average over the three Glucose and three Cadical runtimes in our analysis. Nonetheless, the analysis results remain similar when using only Cadical or using only Glucose runtimes.

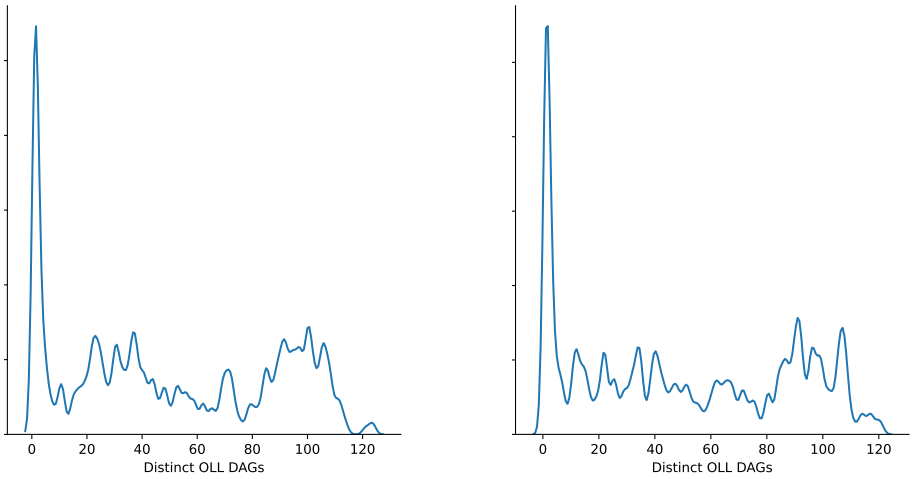
6.1 Variance Analysis

The first step in our analysis is ensuring that we are indeed able to have enough variance in our results. We first remove duplicate OLL DAGs. From the 48 914 gathered OLL DAGs for unweighted instances 21 657 are distinct. For weighted instances the respective numbers are 56 193 and 21 855. Figure 1 shows how these distinct OLL DAGs are distributed over the instances. While we are able to sample several distinct OLL DAGs for most instances, we only obtained a single distinct OLL DAG for 46 unweighted and 46 weighted instances. It is not clear if these instances allow for more than one distinct OLL DAG, as some instances, e.g., instances where we obtain only unit cores, do not allow for different OLL DAGs.

Further, we analyze how much our metrics vary over these distinct OLL DAGs. We normalize our data separately by instance as stated above and then rescale the values such that the same value as the fastest run results in 0, positive values indicate higher values, negative values indicate lower values. Precisely, a value of 1 signifies that the value is double that of the baseline and -1 indicates a halving of the baseline. From there on the values increase such that 2 indicates triple the baseline and -2 one third.

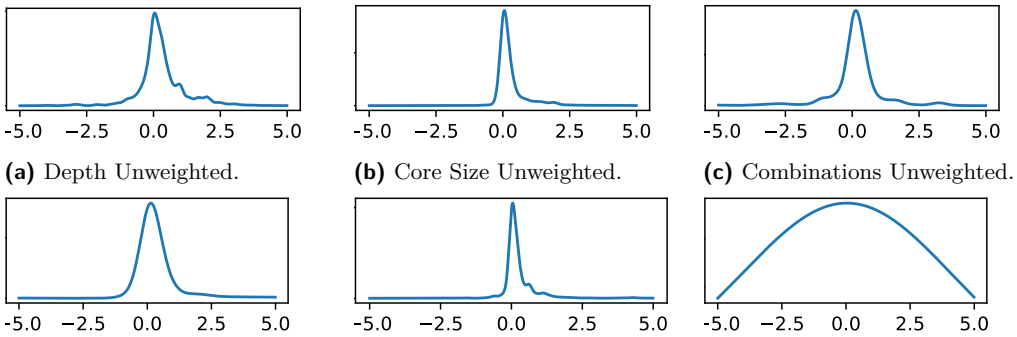
The density plots in Figure 2 show that we are indeed able to sample OLL DAGs with considerable variance in the metric values. In case of the number of combination, the logarithm was used due to the size of the values. The majority of samples has a similar value to the baseline and we have many samples that are within half and double the baseline, often wider. The figures are zoomed in and for all the displayed metrics, we have several outliers as far as ten times or more of the baseline.

[3] We give more results on this in the supplementary material.



(a) Unweighted. **(b)** Unweighted.

Figure 1 Density plots showing the number of distinct OLL DAGs found per instance.



(a) Depth Unweighted. **(b)** Core Size Unweighted. **(c)** Combinations Unweighted.

(d) Depth Weighted. **(e)** Core Size Weighted. **(f)** Combinations Weighted.

Figure 2 Density plots for different metrics for unweighted and weighted instances.

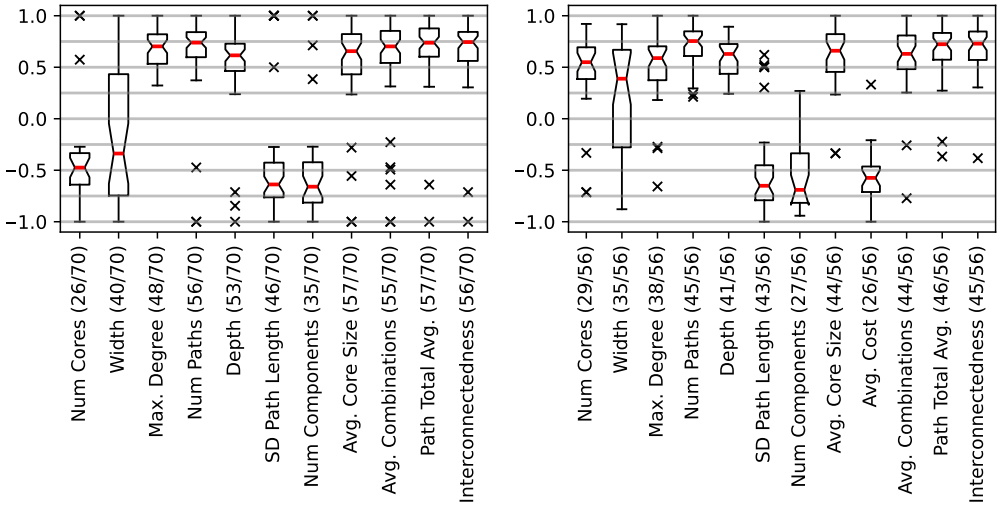
Our results indicate that the heuristics' impact on the metrics is largely as we expected. A notable exception is that disjoint cores does not seem to correlate well with the width of the OLL DAG. However, our experimental setup cannot conclusively confirm this impact, as this would require an ablation study where only configurations varying a single heuristic are used. Due to the already large number of configurations, this was out-of-scope for this paper.

6.2 Runtime Analysis

In this analysis, we ask which metrics correlate well with the runtime. We compute for each metric and instance pair the correlation coefficient. We use the Spearman correlation, as we do not expect a linear correlation, but a rank correlation. Figure 3 shows the results per metric for unweighted and weighted instances respectively[4]. The graph only contains results, where the p-value is smaller or equal than 0.05, the number of instances where the p-value was small enough is indicated in the labels. A high p-value is mainly caused by too

[4] Further boxplots in the appendix show that these results are stable given different parameterizations.

few samples, as too many runs are inconclusive, or too little variation in the metric's value. Further, our hypothesis is that changes in the metric correlate with changes in the runtime. This is slightly different from correlation, as we do expect changes in the runtime, even if the metric does not change. Hence, high correlation confirms our hypothesis, but potentially to a higher degree than the correlation coefficient shows.



(a) Unweighted. **(b)** Weighted.

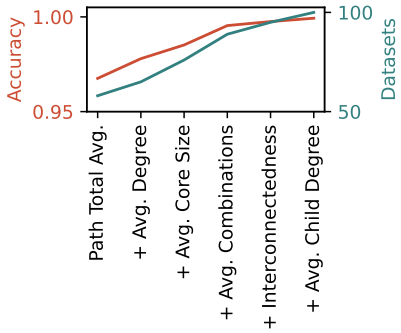
Figure 3 Boxplot of runtime correlations over different instances. The labels shows for how many instances the result was statistically significant, insignificant results are not included.

The results show that high OLL DAG complexity indeed correlates well with high runtime. The best metric is the number of paths in the OLL DAG, as it has a high correlation that does not vary too much from instance to instance and is statistically significant on most instances. Interconnectedness performs almost as well. Surprising is the good correlation for the standard deviation over the longest path lengths among the source-sink paths and the overall inconclusive correlation of the DAG's width.

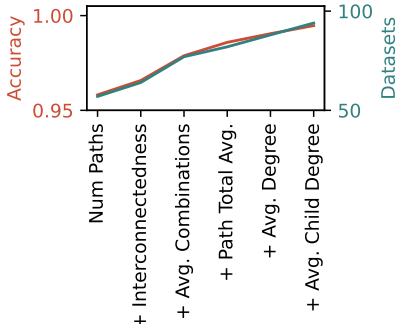
The OLL DAG analysis can also capture important properties that are not directly associated with the graph structure, such as the average number of combinations, which also has a good correlation with the runtime. Further, the average cost tends to correlate with low runtime as is expected. However, many results are statistically not significant and for the remainder the correlation varies a lot.

The negative correlation of the number of cores for unweighted instances is interesting. As the number of cores extracted in a successful run is always the same for unweighted instances, the only reason for this metric to vary is core exhaustion. Therefore, core exhaustion correlates with higher runtime. This does not necessarily indicate that core exhaustion is detrimental to good runtime, but rather that core structures without exhaustible cores might be preferable.

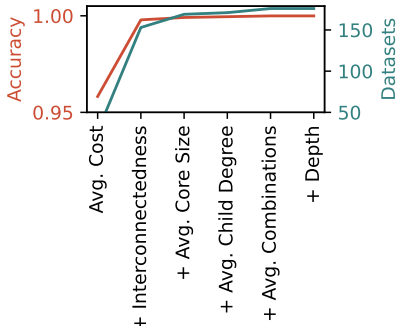
Overall, our results show that the OLL DAG is indeed a good model for the complexity arising from reformulation, as many metrics have a high correlation with the runtime. Selecting cores that result in a simple OLL DAG structure seem to correlate well with low runtime.



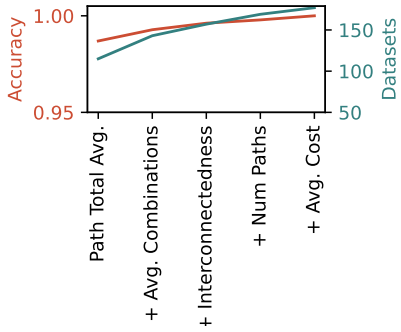
(a) Decision tree/Unweighted.



(b) Decision List/Unweighted.



(c) Decision Tree/Weighted.



(d) Decision List/Weighted.

Figure 4 Constrained models explaining failed runs.

6.3 Reasons for Failure

In our last analysis we want to find indicators for a run failing or succeeding. For this purpose, we analyze the 32 unweighted instances and 23 weighted instances where we have both successful and failed runs. We compare runs on these instances by analyzing the OLL DAGs at the various failures points, i.e., when the runs extracted the same number of cores. We treat each instance and failure point as a separate classification dataset and thereby obtain 101 datasets for unweighted instances and 177 datasets for weighted instances. Each run at a specific failure point is represented by one sample. 87.0% of the 5 667 samples in the unweighted datasets are from successful runs as are 95.2% of the 9 430 samples in the weighted datasets.

The first question we ask is how well any metric on its own can explain the failures. Thus, we analyze how well a single metric can separate all the samples in all datasets into successful and failed samples, i.e., how well we can *split* the datasets using only one metric and a different value per dataset. We measure this using the number of datasets that are fully explained, i.e., after the split we have only successful samples in one half and only unsuccessful in the other. Further, we are interested in the *accuracy*: how many samples agree with the majority class in their half.

For unweighted instances the best metric is the path total average – the average over all nodes of the number of source-sink paths passing that node – explaining 58 datasets and achieving an accuracy of 96.9%. The samples from weighted instances are easier to explain: the average child degree explains 124 datasets and achieves over 99.0% accuracy.

Next, we explore if the combination of metrics can explain more failures. We use specialized decision trees for this purpose, where the decision tree for each dataset has to split on the same sequence of metrics in each branch and can split on each metric at most

once. The decision trees and branches only differ in the values they split on. Further, we also use decision lists, i.e., decision trees that have only one branch with predicate nodes, thereby restricting the model even further. We enforce the same sequence of metrics in every decision tree such that we obtain explanations that apply to all datasets. The respective results are in Figures 4a to 4d.

The datasets from the weighted instances are easier to explain. Although a bad split on its own, average cost in conjunction with metrics measuring the graph complexity can explain almost all failures. This indicates that low average cost might not lead to failures, as long as the OLL DAG remains comparatively simple. Further splits manage to explain all remaining datasets, with decision tree and list performing almost the same. The decision list's splits must separate the samples such that one of the halves contains only a single class, which will become the leaf. Hence, the average cost is not a viable split, and the combination of the path metric and the number of combinations can already explain almost all failures.

The datasets from unweighted instances are much harder to classify correctly. The two models agree mostly on the important metrics in different orders. The number of paths is again very important, followed by in-degree metrics, either the average degree or child degree. The average number of combinations is important as is the interconnectedness, but to a lesser degree than the path and degree metrics. This suggests that a large number of paths and the resulting DAG structure is the best indicator for a failing unweighted run.

We analyze the OLL DAGs of instances with hard to explain failures further, by looking at the instances individually. On those instances, specific combinations of metrics are important. E.g., a large number of cores does not lead to a failure as long as the number of paths stays comparatively low; OLL DAGs with a comparatively large number of paths do not fail, as long as the core sizes are comparatively small and the number of combinations low. Hence, these instances do not offer a simple reason for failures, but require complex explanations. However, the OLL DAG still contains the necessary information. This shows that the explanations provided by the decision models are overall very succinct, as they capture all instances we analyzed, while explanations for single instances may already require three or more metrics as an exhaustive explanation.

7 Conclusion and Future Work

We showed that the OLL DAG is a good abstraction to analyze the impact of different possible reformulations of a MaxSAT instance. This abstraction allowed us to collect a large number of OLL DAGs and analyze why some OLL runs perform well and others do not. This analysis revealed that certain structural properties of the OLL DAG, foremost the number of root-leaf paths, do indeed have a strong correlation with solving performance.

We see several avenues of future research based on OLL DAGs. The first one would be to develop heuristics that specifically target structural properties using and extending our results. Further, we have seen that complex structure correlates with high runtime. This is not always true, as width has a very weak correlation. Hence, further distinguishing which structural complexities correlate with bad performance is of great interest. The large number of duplicate OLL DAGs also shows how difficult it is to sample distinct OLL DAGs. Further methods for varying the OLL DAGs might provide more insights and these methods could also lead to performance improvements. Finally, we are interested in incorporating instance specific properties for a more focused analysis.

References

1. Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, pages 211–221. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.ICLP.2012.211.
2. Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving sat-based weighted maxsat solvers. In Michela Milano, editor, *Principles and Practice of Constraint Programming – 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2012. doi:10.1007/978-3-642-33558-7_9.
3. Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving WPM2 for (weighted) partial maxsat. In Christian Schulte, editor, *Principles and Practice of Constraint Programming – 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 117–132. Springer, 2013. doi:10.1007/978-3-642-40627-0_12.
4. Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. doi:10.1142/S0218213018400018.
5. Fahiem Bacchus and Nina Narodytska. Cores in core based maxsat algorithms: An analysis. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014 – 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 7–15. Springer, 2014. doi:10.1007/978-3-319-09284-3_2.
6. Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 – October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.
7. Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided maxsat solving. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction – CADE 29 – 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2023. doi:10.1007/978-3-031-38499-8_1.
8. Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in sat-based maxsat solving. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming – 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, 2017. doi:10.1007/978-3-319-66158-2_42.
9. Jeremias Berg, Matti Järvisalo, Benjamin Kaufmann, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024. Presented at SAT 2024, 2024. URL: https://maxsat-evaluations.github.io/2024/mse24-talk.pdf.
10. Jeremias Berg, Matti Järvisalo, Benjamin Kaufmann, Ruben Martins, Andreas Niskanen, and Tobias Paxian. MaxSAT Evaluation 2024 : Solver and Benchmark Descriptions. Technical report, University of Helsinki, 2024.
11. Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. Cadical 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification – 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.
12. Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, BMC@CAV 2003, Boulder, Colorado, USA, July 13, 2003*, volume 89(4) of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003. doi:10.1016/S1571-0661(05)82542-3.

13 Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 – March 1, 2017*. The Internet Society, 2017. URL: `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-synthesis-semantic-malware-signatures-using-maximum-satisfiability/`.

14 Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. `doi:10.1007/978-3-319-94144-8_26`.

15 Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient maxsat solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019. `doi:10.3233/SAT190116`.

16 Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 165–176. ACM, 2015. `doi:10.1145/2771783.2771798`.

17 Stepan Kochemazov, Victor Kondratiev, and Irina Gribanova. Empirical analysis of the RC2 maxsat algorithm. In *46th MIPRO ICT and Electronics Convention, MIPRO 2023, Opatija, Croatia, May 22-26, 2023*, pages 1027–1032. IEEE, 2023. `doi:10.23919/MIPRO57284.2023.10159820`.

18 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *J. Mach. Learn. Res.*, 23:54:1–54:9, 2022. URL: `https://jmlr.org/papers/v23/21-0888.html`.

19 João Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010*, pages 9–14. IEEE Computer Society, 2010. `doi:10.1109/ISMVL.2010.11`.

20 João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.*, 62(3-4):317–343, 2011. `doi:10.1007/S10472-011-9233-2`.

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

22 António Morgado, Carmine Dodaro, and João Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming – 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, 2014. `doi:10.1007/978-3-319-10428-7_41`.

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

24 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 193–202. IEEE, 2019. `doi:10.23919/FMCAD.2019.8894273`.

25 Nina Narodytska and Nikolaj S. Bjørner. Analysis of core-guided maxsat using cores and correction sets. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 26:1–26:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.26.

26 Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. Maximum satisfiability in software analysis: Applications and techniques. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification – 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 68–94. Springer, 2017. doi:10.1007/978-3-319-63387-9_4.

27 João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-131.

28 Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. Sat-based techniques for determining backbones for post-silicon fault localisation. In Zeljko Zilic and Sandeep K. Shukla, editors, *2011 IEEE International High Level Design Validation and Test Workshop, HLDVT 2011, Napa Valley, CA, USA, November 9-11, 2011*, pages 84–91. IEEE Computer Society, 2011. doi:10.1109/HLDVT.2011.6113981.