Efficient and Verifiable Proof Logging for MaxSAT Solving

Raoul van Doren
ETH Zurich, Switzerland
rvandoren@student.ethz.ch

Timos Antonopoulos
Yale University, USA
timos.antonopoulos@yale.edu

Ruzica Piskac
Yale University, USA
ruzica.piskac@yale.edu

***Abstract*—MaxSAT solvers are increasingly used as back-ends in software engineering tools. Yet their results have lacked automatically checkable certificates of optimality. While SAT solvers emit DRAT proofs of (un)satisfiability, MaxSAT must additionally prove that no lower-cost solution exists. Existing approaches either cover only isolated solving paradigms or reduce MaxSAT reasoning to heavyweight pseudo-Boolean proofs, yielding impractical verification overhead.**

We present the first MaxSAT-specific proof-logging framework for core-guided OLL solvers. We formalize native inference rules for cores, cliques, hardenings, totalizer updates, and bound adjustments, and implement both a human-readable logger and a compact binary DAG logger in EvalMaxSAT. Evaluation on the 2024 MaxSAT competition dataset confirm the practicality and scalability of our certification pipeline, paving the way for trustworthy, solver use.

***Index Terms*—MaxSAT, proof logging, solver verification, core-guided OLL, debugging**

I. Introduction

Software engineering increasingly relies on SAT solvers [1] for a wide range of automation tasks including program verification, test-suite generation and optimization, configuration management, model synthesis, static analysis and automated program repair [2]–[12]. Modern SAT solvers are powerful and well-engineered tools that can handle millions of variables and tens of millions of clauses, depending on the structure and complexity of the problem. An annual SAT Competition [13] showcases different SAT solvers and plays a crucial role in advancing their development. It provides challenging benchmarks and sets performance standards that push solver efficiency and scalability. However, for many years it was a standard that solvers could simply return "SAT" or "UNSAT" without any justification or proof, which lead to occasional disqualifications in the SAT competitions for incorrect UNSAT claims. To restore trust, the SAT community introduced proof certificates: RUP-style logging in 2003 [14], RAT in 2012 [15], DRUP in 2013 [16], and finally DRAT in 2014 [17]–[19]. Since 2013, all solvers in the SAT Competition's main track must emit a DRAT proof for every UNSAT result [20], ensuring that solver solutions are machine-checkable and reliable in safety-critical and auditable contexts.

While many software engineering tasks may initially appear to be well-suited for SAT encoding, a closer examination often reveals that effective formalization requires going beyond SAT and demands optimization techniques. For instance, tasks such as identifying minimal fault-localization regions in complex debugging scenarios [21], detecting bugs in distributed systems [22], and synthesizing quantum error correction codes [23] can be formulated as SAT problems with an additional requirement to find an optimal solution based on a defined cost metric. These types of problems are more naturally expressed as weighted MaxSAT instances [24], [25].

In the classical SAT problem the goal is to find a solution for a given set of Boolean clauses. All clauses are treated equally and the found model has to be a model for every clause. MaxSAT extends the classical SAT problem by introducing two types of clauses: hard clauses, which must be satisfied, and soft clauses, which should be satisfied, but if they are not this violation incurs a weight-based penalty. The goal is to find a variable assignment that satisfies all hard clauses while minimizing the total weight of the unsatisfied soft clauses. Over the past decade, MaxSAT solver strategies and their performance have drastically increased similarly as for SAT solvers: they also can address the problems with millions of variables and clauses and there is also a yearly competition of MaxSAT solvers [26].

Yet, unlike SAT, where DRAT [17]–[19] proofs have been mandatory in the main track of the SAT Competition since 2013, MaxSAT lacks a single, widely supported proof format that certifies both feasibility and optimality. While a few specialized approaches emit certificates for particular solving portions, there is no general, competition-level standard that everyone uses to demonstrate that no better solution exists. This threatens the exact correctness guarantees that motivated SAT-level certification in the first place.

Extending traditional SAT proof formats, such as DRAT, is insufficient to capture the optimization-specific reasoning required in MaxSAT solving. MaxSAT solvers are working in a core-guided paradigm: each time the solver identifies an unsatisfiable core comprising only of soft literals, it must not only record the literals involved but also the minimum weight among them. This minimum weight is critical and it must be added to the lower bound of the objective function to maintain sound reasoning about optimality. Consequently, a valid MaxSAT proof should log every conflict along with the exact weight update step. Unfortunately, the existing MaxSAT proof efforts trying to solve this problem either target only isolated algorithmic phases [27] or reduce all MaxSAT inferences to general pseudo-Boolean proofs [28] (e.g., VeriPB [29]–[31]), incurring substantial verification overhead and ignoring

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00113

domain-specific optimizations.

In this paper, we close the gap between MaxSAT solving and certification with the first native proof-logging framework. Our system is designed to wrap around any modern MaxSAT solver, based on the OLL algorithm [32]. It is the state-of-the-art core-guided approach used by all top entrants in the 2024 MaxSAT Competition. Our system produces both, a "normal" proof log, in human-readable text form, listing each inference with references to its premises and conclusions, as well as a proof in graph format where nodes represent soft literals and edges encode the dependency and weight-redistribution structure.

Both formats capture the full solver reasoning of the OLL algorithm. Proof sizes stay under 0.1 GB even on our largest benchmarks, showing that our approach is capable of scaling to competition benchmarks. Our contributions are as follows:

- We formalize proof rules for the full OLL workflow.
- We implement both a human-readable normal logger and a compact binary DAG logger within EvalMaxSAT, a top-ranked SOTA OLL solver.
- By leveraging domain knowledge and avoiding translation to pseudo-Boolean proofs, we achieve up to an order of magnitude faster verification than prior approaches, with median verification overhead under 2% of solve time.
- On all 571 benchmarks from the 2024 MaxSAT Competition, our framework successfully verifies every solved instance while producing manageable proof sizes and overhead.

Beyond their theoretical significance, our contributions are highly relevant for software engineering. MaxSAT is already used in diverse SE tasks such as automated program repair, test minimization, fault localization, and configuration optimization [21]–[23], [33]. In these domains, correctness and trust are paramount: an incorrect solver result can propagate into missed faults, unsafe repairs, or misconfigured systems. Certification not only guarantees correctness of results but also enables auditability, post-run analysis and regulatory compliance. For instance, in safety- or security-critical pipelines, certificates can be archived and automatically checked as part of continuous integration, ensuring regressions are caught and decisions are traceable. Moreover, human-readable proof logs support debugging and explainability, allowing developers to understand which constraints drove a decision, rather than treating the solver as a black box.

The remainder of this paper is organized as follows. Section 2 reviews background on OLL-based MaxSAT. Section 3 introduces our proof rules and logging formats. Section 4 describes implementation details and presents the evaluation. Section 6 concludes the work.

II. Background and Motivation

A. Preliminaries on Boolean Formulas and MaxSAT

We denote by x_i a Boolean variable that takes values in $\{0, 1\}$, where 1 reporesents (`true`) and 0 represents (`false`). An assignment for a variable is mapping from a variable to a value $\{0, 1\}$, while a complete assignment is a mapping $\mu : \{x_1, \ldots, x_n\} \to \{0, 1\}$, thereby assigning every variable in a set. A literal is either a variable x_i or its negation $\neg x_i$. A clause $(x_1 \vee x_2 \vee \ldots \vee x_n)$ is the disjunction of its literals, and is satisfied by μ if at least one x_j evaluates to 1 under μ. A formula in conjunctive normal form (CNF) is a conjunction of clauses $\bigwedge_{j=1}^{m} C_j$ and is satisfied by μ exactly when every C_j is satisfied. An unsatisfiable core/conflict is a subset of literals whose conjunction with the hard clauses leads to an unsatisfiable formula. A clique is a set of pairwise-conflicting literals, i.e., each pair leads to an unsatisfiable formula when in conjunction with the hard clauses.

In the MaxSAT problem, each clause C_j is designated hard or soft. Hard clauses must be satisfied, while soft clauses may be violated at a nonnegative integer weight $w_j > 0$. Given a complete assignment μ, the total cost is

$$\text{Cost}(\mu) = \sum_{\substack{C_j \text{ is soft} \\ \mu(C_j)=0}} w_j.$$

and the goal is to find an assignment satisfying all hard clauses while minimizing $Cost(\mu)$.

An empty clause ($\bot$) is always unsatisfied. Thus:

- A hard empty clause leads to unsatisfiable problem instance.
- A soft empty clause of weight w contributes w to the cost of every assignment.

B. Translating Soft Clauses to Weighted Literals

To facilitate the MaxSAT solving process, we transform each soft clause into a weighted unit literal. Let α be the set of hard clauses, β the set of soft literals (initially empty), and γ the set of totalizer literals (initially empty). For each soft clause C_j of weight w_j:

1) Introduce a fresh soft variable $s_j \notin \text{Var}(F)$.
2) Form the extended clause
$$C'_j = C_j \vee \neg s_j,$$
and add C'_j to α.
3) Add s_j to β with weight $w(s_j) \leftarrow w_j$.

Example. A soft clause $(x_1 \vee x_2)$ of weight 3 becomes:

$$(x_1 \vee x_2 \vee \neg s_1) \text{ with } w(s_1) = 3$$

so that either the original clause holds or $s_1 = 0$ incurs cost 3.

C. Cardinality Constraints

A cardinality constraint t restricts how many literals in a set S may be falsified. For integer $k \geq 0$, we require

$$\sum_{x \in S} (1 - \mu(x)) \leq k.$$

To encode this in CNF we make use of a totalizer encoding [34], [35]. The effect is that if more than k literals in S are falsified, then $t = 0$ induces a predefined cost.

D. Preservation of the MaxSAT Objective

Let μ be any assignment to the original variables, the soft literals β, and totalizer literals γ. Then:

- An extended soft clause C_j is satisfied if $\mu(C_j) = 1$ or $\mu(s_j) = 1$. Otherwise $\mu(s_j) = 0$ which incurs cost $w(s_j)$.
- A totalizer literal $t \in \gamma$ is satisfied if its CNF encoding holds. Otherwise $\mu(t) = 0$ incurs cost $w(t)$.

Hence the total cost under μ is

$$\text{Cost}(\mu) = \sum_{s \in \beta} w(s) \cdot \mathbb{1}_{\mu(s)=0} + \sum_{t \in \gamma} w(t) \cdot \mathbb{1}_{\mu(t)=0}$$

which exactly matches the original MaxSAT objective of summing weights of violated soft clauses.

E. Overview of the OLL Algorithm

The OLL MaxSAT algorithm maintains a growing set of soft literals and cardinality constraints and proceeds as follows:

1) **Setup.**
   - Encode all hard clauses in a SAT solver.
   - Transform each soft clause into an extended clause $C \vee \neg s$ and add s (with its weight) to the set of assumable literals.
2) **Iterative Solve.** Repeat until the solver returns SAT:
   a) Assume all current soft and totalizer literals to be true.
   b) Solve the CNF under these assumptions.
   c) If UNSAT, extract an unsatisfiable core S.
   d) Relax S by marking each soft literal in S as paid (incurring its weight) and/or introducing or incrementing a cardinality constraint $\sum_{x \in S} \neg x \leq k$. Add the new totalizer literal (with its weight) to the assumptions.
3) **Result.** When the solver returns SAT, the current model satisfies all hard clauses and minimizes the sum of weights of relaxed literals.

F. Illustrative Example: Abstraction Refinement in Software Verification

Modern software verifiers use predicate abstraction to decrease the complexity of large programs. The abstraction must capture enough behaviors to prove safety, yet it must try to simplify the program as far as possible to be useful.

Consider the following simple program fragment with a single counter and a safety check as shown in Fig. 1.

```
if (x > 10) {
  y := y + 1;
} else {
  y := y - 2;
}
assert(y >= 0);
```

Fig. 1: Example code.

A predicate-abstraction verifier must choose which predicates to track in the Boolean model so that any violation of the final assertion is detected. Each predicate p_i is a simple Boolean test which represents the abstract properties that we would like to track in the program verifier. However, we also assume that each predicate has a predefined "omission-cost": if we decide not to track p_i, we pay a penalty that makes the Boolean formula easier but increases the risk of missing an unsafe behavior. Suppose our candidate predicates are

$$p_1 : x > 10, \quad p_2 : y \geq 0, \quad p_3 : y > 1.$$

To decide which predicates to actually track, we introduce a fresh Boolean soft literal s_i for each predicate p_i. Semantically,

$$s_i = 0 \quad \text{means "we track } p_i\text{" (i.e., no penalty),}$$
$$s_i = 1 \quad \text{means "we omit } p_i\text{" (i.e., incurs cost).}$$

In each hard-clause implication, we then extend the predicate test by $\neg s_i$, so that omitting p_i (setting $s_i = 1$) causes the implication to hold.

Each p_i carries an omission cost

$$w(p_1) = 10, \quad w(p_2) = 8, \quad w(p_3) = 2,$$

so setting $s_i = 1$ omits p_i at cost $w(p_i)$. Because $w(p_3) = 2$ is relatively small, omitting p_3 is "cheap", whereas omitting p_1 or p_2 would be more expensive.

To guarantee safety, we must ensure that whenever the concrete execution could make `assert(y >= 0)` fail, the abstract model has enough predicates to distinguish that behavior. Concretely:

1. On the `then`-branch ($x > 10$), if p_2 were false ($\neg p_2$), the assertion could fail. Thus we require

$$(p_1 \wedge \neg p_2) \longrightarrow p_2,$$

which in CNF becomes the hard clause

$$(\neg p_1 \vee p_2).$$

2. On the `else`-branch ($\neg(x > 10)$), a stronger check is needed: if p_2 alone does not suffice, the more precise predicate p_3 may be required. That yields

$$(\neg p_1 \wedge \neg p_2) \longrightarrow (p_2 \vee p_3),$$

i.e. the hard clause

$$(p_1 \vee p_2 \vee p_3).$$

To build the MaxSAT encoding:

- We add the two hard clauses $(\neg p_1 \vee p_2)$ and $(p_1 \vee p_2 \vee p_3)$ that must be satisfied.
- We then add soft clauses $(\neg s_i)$ of weight $w(p_i)$. In each hard clause, we replace each p_i (or $\neg p_i$) by the disjunction $(p_i \vee s_i)$ (or $(\neg p_i \vee s_i)$). This replacement is applied simultaneously to every occurrence of each p_i in a hard clause. For example, the hard clauses become:

$$(\neg p_1 \vee p_2) \quad \longrightarrow \quad (\neg p_1 \vee p_2 \vee s_1 \vee s_2),$$
$$(p_1 \vee p_2 \vee p_3) \quad \longrightarrow \quad (p_1 \vee p_2 \vee p_3 \vee s_1 \vee s_2 \vee s_3).$$

Now, if we choose to omit p_3 by setting $s_3 = 1$, then the second clause

$$(p_1 \vee p_2 \vee p_3 \vee s_1 \vee s_2 \vee 1) = \texttt{true},$$

so that clause no longer enforces any relationship among p_1, p_2. However, as long as we desire to track p_1 and p_2 (i.e., $s_1 = s_2 = 0$), the first clause is enforced.

A MaxSAT solver might, for example, omit both p_2 and p_3 (paying cost $8+2=10$) and still satisfy the transformed hard clauses, even though omitting only p_2 (cost 8) would have been optimal. Without a machine-checkable proof of optimality, such suboptimal assignments could escape detection. In safety-critical verification pipelines, an incorrect relaxation or miscalculated weight may yield an abstraction that misses a real bug or falsely reports safety.

Our certification framework ensures that every relaxation, core extraction, and clique formation step is logged and later re-verified. This way, the verifier confirms not only feasibility but also optimality of the chosen abstraction, preventing incorrect tradeoffs in predicate omission. Such verifiable guarantees are essential for reliable automated verification in practice, where tools must be trusted in continuous integration settings and regulatory audits [36], [37].

III. Related Work

Our work concentrates exclusively on the core-guided procedure known as OLL [32], which underlies the top three [38]–[40] MaxSAT solvers in the 2024 MaxSAT Competition [26]. As the empirical leader among all MaxSAT techniques, OLL forms the foundation of the current state of the art MaxSAT solvers.

Despite rapid advances in MaxSAT algorithms, proof logging lags behind. One might ask whether existing SAT proof systems (e.g., DRAT [17]–[19]) could suffice for MaxSAT. In practice, however, DRAT's clause-only language cannot express reasoning about optimization objectives, and a single certifying call to a SAT solver at the end of the solving process cannot establish the optimality of the solution.

Modern MaxSAT solvers rely on four primary paradigms, Branch-and-Bound (BnB) [41], Solution-Improving Search [42]–[44], Implicit Hitting Set (IHS) [45], [46] and Core-Guided Search [47]–[50], each applying its own inference patterns to determine a solution. This diversity hinders defining a unified proof logic. Existing proof-logging efforts target only individual aspects of the problem [51]–[60], such as certifying preprocessing techniques [27], automatic proof generation for the unweighted linear SAT-UNSAT solver QMaxSAT [61], and a high-level proof sketch for OLL-style core-guided MaxSAT [32] without automatic proof generators. More recent work utilizes the pseudo-Boolean checker VeriPB [29]–[31] to verify core-guided solvers, translating the solution inference into a 0–1 linear-inequality proof [28]. Although more expressive than DRAT, this approach treats each MaxSAT step as a PB inference and cannot exploit fast SAT-level RUP checks, proof trimming, or binary formats. As a result, verification often dominates solver runtime and misses domain-specific optimizations.

As the de facto state of the art, OLL-based techniques warrant a dedicated certification framework. To our knowledge, the only prior attempt to certify OLL-based MaxSAT solving is the `certified-cgss` tool [62]–[64], which reduces solver reasoning to pseudo-Boolean proofs for verification with VeriPB. While correct, this reduction sacrifices MaxSAT-specific invariants and relies entirely on the soundness of the translation and of VeriPB itself. The resulting certificates are large, hide the solver's native reasoning, and do not give the possibity to exploit SAT-level optimizations such as DRAT trimming or efficient clause hashing.

In contrast, our framework defines inference rules directly within the OLL workflow (core extraction, clique relaxation, conflict redistribution), producing both human-readable logs and compact DAGs with shared subproofs. This design avoids translation overhead, keeps certificate sizes manageable, and exposes solver logic in a transparent form that is useful for debugging, explainability, and integration into software engineering pipelines. We further support optional DRAT proofs for UNSAT calls, allowing our verifier to leverage mature SAT-checking optimizations that PB-based systems cannot exploit. Overall, our specialized design bridges the gap between solver reasoning and efficient certification, providing both theoretical soundness and practical usability.

IV. Proof-Logging System

In the following chapter, we construct a proof system for the MaxSAT OLL algorithm and prove its soundness and completeness.

A. Proof States

A proof state is represented as a quintuple:

$$F = \langle \alpha, \beta, \gamma, L, U \rangle,$$

where:

- α is the set of hard clauses, which must be satisfied,
- β is the set of soft literals,
- γ is the set of totalizer literals,
- L is the current lower bound on the optimal solution cost,
- U is the current upper bound on the optimal solution cost.

The initial proof state is defined as:

$$F_0 = \langle \alpha_0, \beta_0, \gamma_0, L_0, U_0 \rangle,$$

with $\gamma_0 = \emptyset$, $L_0 = 0$, and $U_0 = \infty$.

B. Inference Rules

A proof rule defines a valid transformation from one proof state to another. Given a current proof state F, a rule produces an updated proof state F' by modifying a subset of components in F. Any component not modified by the rule remains unchanged in F'.

The goal is to reach a final proof state:

$$F_{\text{final}} = \langle \alpha, \beta, \gamma, L, U \rangle \quad \text{where} \quad L \geq U.$$

At this point, we have a concrete value on the optimal cost, and the proof concludes.

Empty Soft Clause (ESC). If a soft clause is empty, it is unsatisfiable in any assignment. Thus, its cost must be paid in any optimal solution and it can be removed from the proof state.

$$\frac{C = \{\} \quad C \in \beta}{L' := L + w(C), \quad \beta' := \beta \setminus \{C\}} \quad \text{[ESC]}$$

Hardening (H). If the cost of a literal is high enough that assuming it is unsatisfied would exceed the current upper bound, then it must be satisfied in any optimal assignment. Thus, it can be moved to the set of hard clause which are required to be satisfied by definition.

$$\frac{l \in (\beta \cup \gamma) \quad w(l) \geq U - L}{\alpha' := \alpha \cup \{l\}, \quad \beta' := \beta \setminus \{l\}, \quad \gamma' := \gamma \setminus \{l\}} \quad \text{[H]}$$

Unsatisfiable Literal (UL). If a soft or cardinality literal leads to an unsatisfiable formula when assumed to be true, it contradicts the current set of hard clauses. Therefore, we add its negation to the hard clauses and increase the lower bound by its weight.

$$\frac{l \in (\beta \cup \gamma) \quad \text{solve}(\alpha \cup \{l\}) = \text{UNSAT}}{\alpha' := \alpha \cup \{\neg l\}, \quad L' := L + w(l), \quad w(l) := 0} \quad \text{[UL]}$$

Clique Relaxation (ClR). If a set of literals forms a mutual conflict (i.e., any pair leads to UNSAT when assumed true simultaneously), then at most one of them can be satisfied at a time. This concept is known as a clique. Cliques are crucial in avoiding quadratic blowup in MaxSAT solving. If k literals are mutually conflicting, encoding all $\binom{k}{2}$ pairwise relaxations would be expensive. Instead, solvers introduce a clique constraint that relaxes them collectively, maintaining soundness while reducing overhead. In our certification setting, we do not reconstruct how the clique was detected, we only verify that the final set indeed represents a valid clique. We can extract the minimum weight from this clique, increase the lower bound accordingly, and introduce a new variable to encode the residual constraints.

$$\frac{\begin{array}{c} C = \{l_1, \ldots, l_n\} \subseteq (\beta \cup \gamma) \\ \forall i \neq j \in |C| : \text{solve}(\alpha \cup \{l_i, l_j\}) = \text{UNSAT} \\ w_{\min} = \arg\min_{l \in C}\{w(l)\} \end{array}}{\begin{array}{c} L' := L + w_{\min} \cdot (|C| - 1) \\ \forall l \in C : w(l) := w(l) - w_{\min} \\ l_f := \text{freshVar}() \\ \beta' := \beta \cup \{l_f\}, \quad w(l_f) := w_{\min} \\ \alpha' := \alpha \cup \{(l_1 \vee \cdots \vee l_n \vee \neg l_f)\} \end{array}} \quad \text{[ClR]}$$

Conflict Relaxation (CoR). Given a set of literals C that cause a conflict (i.e., $\text{solve}(\alpha \cup C) = \text{UNSAT}$), we perform a weight redistribution and introduce a totalizer constraint to ensure proper relaxation.

$$\frac{\begin{array}{c} C = \{l_1, \ldots, l_n\} \subseteq (\beta \cup \gamma) \\ \text{solve}(\alpha \cup C) = \text{UNSAT} \\ w_{\min} = \arg\min_{l \in C}\{w(l)\} \end{array}}{\begin{array}{c} L' := L + w_{\min} \\ \forall l \in C : w(l) := w(l) - w_{\min} \\ t_{(\{\neg l_1, \ldots, \neg l_n\}, 1)} := \text{freshVar}() \\ t_{(\{\neg l_1, \ldots, \neg l_n\}, 1)} \Leftrightarrow \sum_{l_i \in C} \neg l_i \leq 1 \\ \gamma' := \gamma \cup \{t_{(\{\neg l_1, \ldots, \neg l_n\}, 1)}\}, \quad w(t) := w_{\min} \end{array}} \quad \text{[CoR]}$$

Totalizer Update (TU). If a totalizer literal with weight 0 does not contribute to the objective anymore, we relax its "at-most" bound by 1 (from k to $k+1$) and assign a new totalizer literal to this updated cardinality constraint with the original weight.

$$\frac{\begin{array}{c} t_{(\{\neg l_1, \ldots, \neg l_n\}, k)} \in \gamma \quad w(t_{(\{\neg l_1, \ldots, \neg l_n\}, k)}) = 0 \\ w_{\text{init}}(t_{(\{\neg l_1, \ldots, \neg l_n\}, k)}) = w \end{array}}{\begin{array}{c} t_{(\{\neg l_1, \ldots, \neg l_n\}, k+1)} := \text{freshVar}() \\ t_{(\{\neg l_1, \ldots, \neg l_n\}, k+1)} \Leftrightarrow \sum_{l_i \in C} \neg l_i \leq k + 1 \\ \gamma' := \gamma \cup \{t_{(\{\neg l_1, \ldots, \neg l_n\}, k+1)}\} \\ w(t_{(\{\neg l_1, \ldots, \neg l_n\}, k+1)}) = w \\ w_{\text{init}}(t_{(\{\neg l_1, \ldots, \neg l_n\}, k+1)}) = w \end{array}} \quad \text{[TU]}$$

Upper Bound Update (UBU). If we find a satisfying assignment μ whose cost is strictly less than the current upper bound, we update the upper bound accordingly.

$$\frac{\begin{array}{c} \text{SAT solver returns model } \mu \\ \text{Cost}(\mu) < U \end{array}}{U := \text{Cost}(\mu)} \quad \text{[UBU]}$$

C. Advanced Proof-Logging via Dependency DAG

In the advanced proof system, we represent an entire OLL derivation as a directed acyclic graph (DAG) whose vertices correspond to soft and totalizer literals, both the original ones and those introduced by relaxation rules, and whose edges record the dependencies and weight-redistributions imposed by clique and conflict relaxations. Each vertex v holds the following entries:

- a unique identifier (in insertion order),
- a type tag (Soft, Hardening, Clique, Conflict, or UnsatLit),
- an initial weight $w_0(v)$,
- a current weight $w(v)$,
- a set of successor literals $C(v) \subseteq V$ for Clique and Conflict nodes,
- a bound parameter $k(v)$ for Conflict nodes.

The DAG is built incrementally. Initially, for each original soft clause we create a soft node v with $w_0(v) = w(v)$ set to the clause weight. Its outgoing edges point to all variables inside of the soft clause.

Whenever a clique relaxation is performed on a core $C = \{u_1, \ldots, u_n\}$, we add a new Clique node v with

$$k(v) = 1, \quad w_0(v) = w(v) = \min_{u \in C} w(u),$$

and add directed edges $v \to u$ for each $u \in C$, simultaneously subtracting $w_0(v)$ from each $w(u)$.

Similarly, for a general conflict relaxation on C, we add a Conflict node v with

$$k(v) = 1, \quad w_0(v) = w(v) = \min_{u \in C} w(u),$$

and add the edges $v \to u$ for all $u \in C$ while reducing each $w(u)$ by $w_0(v)$.

Finally, each hardening or unsatisfiable literal inference is

recorded as a Hardening or UnsatLit node with one outgoing edge to its affected literal.

Once the DAG construction is complete, verification proceeds in reverse topological order. We initialize the verifier's lower bound L to the solver's final optimum U_{Final}. Then, for each node v in reverse insertion order, we perform the following checks and updates:

Soft nodes. We verify that $w_0(v)$ matches the original clause weight. If they do not match, the proof is invalid. No change to the bound or to other weights is performed.

Hardening nodes. Let u be the unique successor of v. We check that

$$L + w_0(u) \;\geq\; U_{final},$$

ensuring that falsifying u would push the cost above the optimum. If this does not hold, the proof is invalid. Otherwise, we add the literal represented by the node u to the set of hard clauses.

Clique nodes. If $C(v) = \{u_1, \ldots, u_n\}$ and $w_0(v)$ is its weight, we first restore the bound

$$L := L - (|C(v)| - 1) \cdot w_0(v)$$

to "undo" the original clique relaxation cost. We then restore each child's weight $w(u_i) := w(u_i) + w_0(v)$. Finally, we add the clique constraint encoded as soft clause to the set of hard clauses.

Conflict nodes. For a Conflict node v with child set $C(v)$ and $k(v) = 1$, we restore

$$L := L - w_0(v) \quad \text{and} \quad w(u) := w(u) + w_0(v) \quad \forall u \in C(v).$$

We also add the cardinality constraint encoded using totalizers to the set of hard clauses. Nodes with $k(v) \neq 1$ (from totalizer updates) are skipped.

UnsatLit nodes. Let u be the child of v. We decrement

$$L := L - w_0(u)$$

and add $\neg u$ to the reconstructed hard clause set.

After all nodes have been processed, a valid proof satisfies $L = 0$ and yields an assignment to the original soft literals whose cost equals U_{Final}. Throughout this verification we also ensure that each clique, conflict, and unsatisfiable-literal inference was extracted in a context consistent with the set of hard clauses at the time of its introduction.

V. Formal Verification

A. Soundness

Observation 1. *To ensure the soundness of the proof system, it must be guaranteed that:*

- *The lower bound L does not overestimate the optimal cost,*
- *The upper bound U does not underestimate the optimal cost.*

For all proofs, we assume the existence of an optimal assignment $\mu^* : l \to \{0, 1\}$. The cost of this assignment is defined as:

$$\text{Cost}(\mu^*) = L + \sum_{l \in \beta} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in \gamma} w(l) \cdot \mathbb{1}_{\mu^*(l)=0}.$$

Theorem 1 (Soundness of the Proof System)**.** *The proof system is sound if at every proof state $F_i = \langle \alpha_i, \beta_i, \gamma_i, L_i, U_i \rangle$ in the proof derivation from F_0 to F_{Final}, the soundness invariant $L_i \leq \mathit{Cost}(\mu^*) \leq U_i$ (see Obs. 1) holds. Thus, if $L_{Final} = \mathit{Cost}(\mu^*) = U_{Final}$, the proof is correct and the solution is proven to be optimal.*

Proof Sketch. Proof by induction:

- **Base Case:** In the initial state F_0, we $L_0 = 0 \leq \text{Cost}(\mu^*) \leq U_i = \infty$. Thus the invariant trivially holds.
- **Step Case:** Suppose the invariant holds in F_i. If we apply any of the previously introduced rules, the new values L and U still satisfy $L \leq \text{Cost}(\mu^*) \leq U$, due to each rule preserving the soundness invariant.
- **Conclusion:** In the final state $F_{Final} = \langle \alpha_F, \beta_F, \gamma_F, L_F, U_F \rangle$, the inequalities $L_F \leq \text{Cost}(\mu^*) \leq U_F$ still apply, thereby completing the soundness proof if $L_F = \text{Cost}(\mu^*) = U_F$.

□

Soundness Proof for ClR. To illustrate how the soundness invariant is preserved by individual rules, we provide a proof for the Clique Relaxation (ClR) rule. Therefore, we have to show that the soundness invariant $L \leq \text{Cost}(\mu^*) \leq U$ still holds after applying the ClR rule.

Proof. We know show that at most one literal from the clique C can be true in any optimal assignment, due to mutual pairwise conflicts. Therefore, at least $(|C| - 1)$ literals in C must be false in any assignment μ^*. So we must add at least $(|C| - 1) \cdot w_{\min}$ to the cost, which justifies the increase of the lower bound L. Assuming the current proof state is valid, we have:

$$\begin{aligned}
& L \leq L + \sum_{l \in \beta} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in \gamma} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \leq U \\
\Leftrightarrow \quad & L \leq L + \sum_{l \in (\beta \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in (\gamma \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \\
& \quad + \sum_{l \in C} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \leq U \\
\Leftrightarrow \quad & L \leq L + \sum_{l \in (\beta \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in (\gamma \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \\
& \quad + \sum_{l \in C} (w(l) - w_{min}) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in C} w_{min} \cdot \mathbb{1}_{\mu^*(l)=0} \\
& \quad \leq U
\end{aligned}$$

We subtract $w_{\min}$ from each literal in C, and introduce a fresh literal l_f with cost $w_{\min}$ to preserve this residual cost. The clause $(l_1 \vee \cdots \vee l_n \vee \neg l_f)$ ensures that if all l_i in C are false, then l_f must also be false and its cost must be added. If

at least one l_i is true, the clause is satisfied and l_f is set to true, avoiding extra cost. Thus, we can rewrite the cost function to:

$$\begin{aligned} L \leq L &+ \sum_{l \in (\beta \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in (\gamma \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \\ &+ \sum_{l \in C} (w(l) - w_{min}) \cdot \mathbb{1}_{\mu^*(l)=0} + (|C| - 1) \cdot w_{min} \\ &+ w_{min} \mathbb{1}_{\forall l \in C:\, \mu^*(l)=0} \leq U \\ \Leftrightarrow \quad L \leq L &+ (|C| - 1) \cdot w_{min} + \sum_{l \in (\beta \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} \\ &+ \sum_{l \in (\gamma \setminus C)} w(l) \cdot \mathbb{1}_{\mu^*(l)=0} + \sum_{l \in C} (w(l) - w_{min}) \\ &\cdot \mathbb{1}_{\mu^*(l)=0} + w_{min} \mathbb{1}_{\mu^*(l_f)=0} \leq U \end{aligned}$$

Note that $\mu^*(l_f) = 1$ if at least one l_i is true, which makes $w_{\min} \cdot \mathbb{1}_{\mu^*(l_f)=0} = 0$. If all l_i are false, then l_f must be false due to the added hard clause, and its cost must be paid. Thus, the validity of the cost function for an optimal assignment is preserved. □

B. Completeness

Theorem 2 (Completeness of the Proof System)**.** *The proof system is complete if the final proof state $F_{Final} = \langle \alpha_F, \beta_F, \gamma_F, L_F, U_F \rangle$ always implies $L_F = U_F$.*

Proof Sketch. We proceed by induction on the proof steps. Initially, the invariant $L_0 \leq \text{Cost}(\mu^*) \leq U_0$ holds trivially. At each step, either (1) a better solution is found, decreasing U, or (2) a conflict is analyzed, increasing L. Each rule preserves the invariant. Since L increases and U decreases monotonically and are bounded, they eventually converge. By soundness, $L = \text{Cost}(\mu^*) = U$ in the final state, proving completeness. □

C. Formal Verification of the Advanced Proof-Logging System

We adopt the same soundness invariant as for the previous proof system: At every point during DAG-based verification, the reconstructed lower bound L satisfies

$$0 \leq \text{Cost}(\mu^*) - L,$$

where $\text{Cost}(\mu^*) = U_{Final}$ is the optimal cost reported by the solver. Equivalently,

$$0 \leq L \leq \text{Cost}(\mu^*).$$

Theorem 3 (Soundness of the Advanced Proof Verification)**.** *If the DAG-based check completes with $L = 0$, then $L_{Final} = Cost(\mu^*) = U_{Final}$ holds and we therefore know that the solver has found an optimal MaxSAT solution μ^*.*

Proof Sketch. We verify that the solution μ returned by the solver is valid and satisfies $\text{Cost}(\mu) = U_{Final}$. The DAG is traversed in reverse, and at each node the cost initially added to the lower bound L is subtracted. If the traversal completes with $L = 0$, this implies that all lower-bound increments have been correctly reversed, and thus $L_{Final} = U_{Final}$. By soundness, this guarantees $\text{Cost}(\mu^*) = U_{Final}$, proving optimality. □

Theorem 4 (Completeness of the Advanced Proof Verification)**.** *If the OLL algorithm reaches a final state with an optimal cost $L = Cost(\mu^*) = U$, then the DAG generated by recording its inferences admits a reverse cost-update check ending with $L = 0$.*

Proof Sketch. Each lower bound update in the OLL algorithm corresponds to a DAG node and reflects a strictly positive cost increase. If the solver terminates with $L = \text{Cost}(\mu^*) = U$, then the total lower bound equals the sum of all updates. Reversing the DAG subtracts each increment exactly once, reducing L back to 0. Hence, the reverse verification succeeds whenever the solver finds an optimal solution, proving completeness. □

D. Illustrative Example

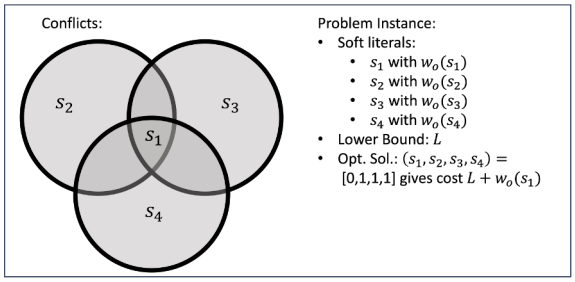

Fig. 2: A visualization of the initial hard and soft clauses before any core relaxation.

Let us demonstrate the procedure followed by the previously described proof loggers. Consider a MaxSAT instance (as shown in Fig. 2) with three unsatisfiable cores of soft literals:

$$\text{core 1: } \{s_1, s_2\}, \quad \text{core 2: } \{s_1, s_3\}, \quad \text{core 3: } \{s_1, s_4\}.$$

We assume that the following relationship between the weights of the soft literals hold:

$$\begin{aligned} &w(s_2) < w(s_1), \quad w(s_3) < w(s_1), \quad w(s_4) < w(s_1), \\ &w(s_1) < w(s_2) + w(s_3) + w(s_4). \end{aligned}$$

We want to show that the lower bound we derive via successive core relaxations (or equivalently clique relaxations for pairs of literals) will match the optimal cost. Concretely, we will show the final cost equals

$$L_0 + w_{\text{old}}(s_1),$$

where L_0 is the lower bound before the problem start and $w_{\text{old}}(\cdot)$ denotes the the initial weight of s_1.

Informal Reasoning. From the problem setup, resolving all three unsatisfiable cores most cheaply requires paying for s_1 only. Intuitively, since $w(s_1)$ is smaller than the sum of the three other literal weights $w(s_2) + w(s_3) + w(s_4)$, forcing s_1 to be false (and thus incurring its cost) is cheaper than any alternative combination of paying for s_2, s_3 and s_4.

Formal Reasoning (Sketch). We simulate the solver's steps across the three cores using successive relaxations:

- **Core 1:** Relax $\{s_1, s_2\}$ by subtracting $w(s_2)$, increasing L by $w(s_2)$ and reducing $w(s_1)$.

- **Core 3:** Relax $\{s_1, s_4\}$ by subtracting the new minimum, either $w(s_4)$ or the updated $w(s_1)$, updating L and adjusting weights accordingly.
- **Core 2:** Finally, relax $\{s_1, s_3\}$ using the remaining weight on s_1 or s_3, again increasing L. Figure 3 shows the problem state after executing the relaxation rule on the core 2.

In all cases, the total increase in the lower bound is exactly $w_{\text{old}}(s_1)$, and the relaxation process ensures this amount is fully accounted for through literal cost redistribution. Thus, the updated $L = L_0 + w_{\text{old}}(s_1)$ matches the cost of the optimal solution where s_1 is violated and all others are satisfied.

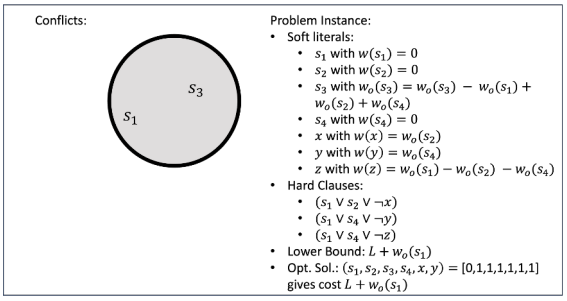

Fig. 3: Final result showing the updated set of clauses and lower bound for the case 1.

Figure 4 highlights how our DAG-based proof logging represents the complete relaxation history. On the left, each unsatisfiable core is encoded as a conflict node pointing to the relevant soft literals with their initial weights. On the right, after successive relaxations, the updated weights and lower bound are shown. The edges record how weight was redistributed during each inference step, ensuring that the verifier can replay the solver's reasoning exactly. This visualization clarifies how the proof DAG compactly encodes dependencies across multiple cores while preserving enough structure to support efficient re-checking. For SE practitioners, such diagrams as defined in the proof log of DAG-proof logger also serve as an explainability aid: they illustrate which assumptions were forced, why certain costs were incurred, and how the solver reached its optimal conclusion.

VI. Evaluation

A. Implementation

We have integrated both the normal and DAG-based proof-logging backends into EvalMaxSAT, one of the three top-ranked OLL solvers in the 2024 MaxSAT Competition. EvalMaxSAT refines each unsatisfiable core via core minimization before applying the relaxation rules. However, this does not impact proof logging as our loggers only record the minimized cores.

The normal logger emits a sequence of entries to a plaintext file. Each entry introduces a new clause or soft literal (hard clause, extended soft clause, totalizer, bound, etc.) and assigns it a unique identifier. The proof generator then emits inference rule in the order in which they occurred and adds its premises and conclusion by referencing to the corresponding identifiers.

The DAG-based logger maintains an DAG of proof states throughout solving and serializes it in a compact binary format after the solving process has concluded. Each node includes all the information required for checking the validity of the corresponding node type.

EvalMaxSAT relies on an external SAT solver and on SCIP for intermediate integer-optimization calls. When the SAT solver returns UNSAT under assumptions, we record its assumptions and optionally request a DRAT proof. If SCIP produces the final optimal assignment, we store the current proof state. The verifier later checks the validity of the stored intermediate proof state and tries to reproduce the optimum starting from the recorded intermediate proof state by performing a black-box SCIP invocation to confirm the claimed optimum.

As noted above, our implementation could be extended to emit a DRAT proof for every conflict extraction. In principle, this would allow the verifier to check each core or clique inference entirely via SAT-level reasoning. In practice however, generating and storing a DRAT proof for every conflict is not scalable: for example, verifying a clique of size n would require $O(n^2)$ pairwise proofs. Instead, we resolve each conflict during verification by performing black-box SAT calls for each conflict which allows us to validate cores and cliques more efficiently. This part of the verification is also parallelizable.

B. Benchmarks and Hardware

Benchmarks. We evaluate our implementations on the Weighted exact MaxSAT track of the 2024 MaxSAT Competition [26], comprising 571 instances drawn from diverse domains such as:

- Preprocessing for Nonmonotonic c-Inference via Partial MaxSAT
- MaxSAT Encodings for Pareto-Optimal Interpretations of Black-Box Models
- Incremental MaxSAT from Train Scheduling Discretizations
- Learning Balanced Rules via MaxSAT
- Minimizing Pentagons in the Plane

Hardware Platform. All experiments run on a single socket Intel Xeon E3-1284L v4 (@2.90 GHz), with 128 GB RAM and Ubuntu 22.04. We allocate 16 GB of RAM per solver invocation.

Timeouts and Memory. We give a 3 600s timeout with 16 GB RAM for the solving and logging process of each benchmark. The verificaiton process of each benchmark is bounded to 7 200s and 16 GB RAM.

Comparison. To compare against prior work, we also evaluate "certified-cgss" [62], the only other proof-logging OLL solver, which emits pseudo-Boolean certificates for VeriPB [29]–[31]. We run it on the same 571 benchmarks under identical constraints.

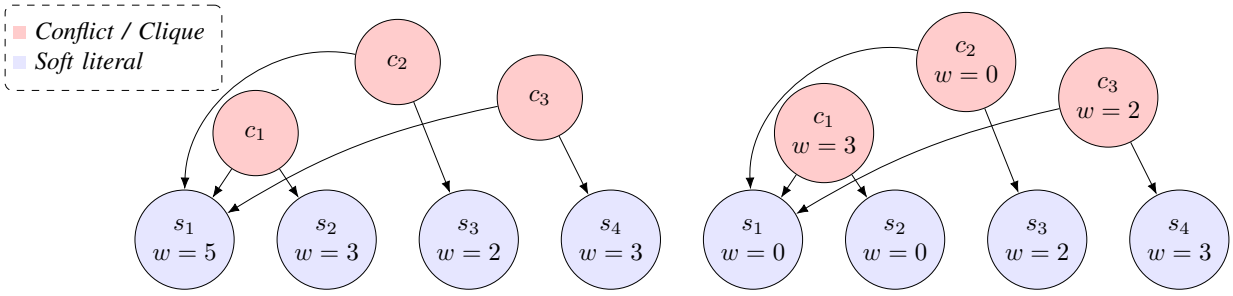

Fig. 4: DAG representation of the illustrative example. Conflict nodes (red) point to their participating soft literals (blue). Weights are updated across relaxations, and the cumulative lower bound increases accordingly. This structure ensures that verification can be performed by a reverse traversal, making solver reasoning transparent and machine-checkable.

C. Research Questions

RQ1: Proof Logging Overhead. How much additional time does proof logging add to plain MaxSAT solving in each mode?

RQ2: Certificate Size. How large are the "standard" proofs versus the binary DAG proofs, and how do they compare to certificates produced by the certified-CGSS solver?

RQ3: Verification Cost. What is the end-to-end time to check each proof, and how does it compare across the two logging modes and to the certified-CGSS baseline?

D. Metrics and Procedure

To understand how proof-logging overhead varies with problem sizes, we group the 435 solved instances into quartiles by three measures: number of hard clauses (#HC), number of soft clauses (#SC), and number of variables (V). Tables I, III and V summarize the advanced (DAG) mode, while Tables II, IV and VI report the corresponding standard mode. In each table:

- **FC:** number of test files in the bin,
- **R(s):** median runtime of the solver without proof logging (baseline),
- **TR(s):** median runtime of the solver with proof logging,
- **IR:** median percentage increase $(\mathrm{TR}-\mathrm{R})/\mathrm{R}\times 100$, with 95% bootstrap confidence intervals (CI),
- **VR:** median verification time (seconds),
- **PS:** median proof size (GB),
- **MW:** median maximum weight,
- **SW:** median sum of weights,
- **OC:** median optimal cost.

Note that an IR value close to 0 indicates negligible runtime overhead. We report the median IR values together with 95% bootstrap confidence intervals, since the mean can be heavily skewed by outliers (e.g., when the baseline runtime R is very small). Because modern MaxSAT solvers rely on randomized heuristics, the logged run may occasionally finish faster than the baseline, which might again lead to negative IR values following from solver nondeterminism. Such negative values should be interpreted as statistical noise rather than a real performance gain.

Table VII summarizes verification performance on the 668 benchmarks solved by both EvalMaxSAT and the certified-cgss solver under identical time and memory limits. We report:

- **VO_EMS** (Verification Overhead for EvalMaxSAT): The ratio of verification time to solve time for our normal and DAG modes.
- **VO_CGSS** (Verification Overhead for certified-cgss): The corresponding slowdown ratios for the certified-cgss tool.
- **VSP** (Verification Speedup): The element-wise ratio $\mathrm{VO\_CGSS}/\mathrm{VO\_EMS}$, indicating how many times faster our verifier runs relative to certified-cgss.
- **PSD** (Proof-Size Decrease): The ratio of EvalMaxSAT proof size to certified-cgss proof size.

All metrics in this table are given as median and 90th-percentile.

RQ1: Proof Logging Overhead

We measure proof-logging overhead by comparing the solver's baseline run time R against its logged run time TR under both normal (standard) and advanced (DAG) modes.

Across all quartiles, the standard logger shows a negligible median overhead: overall its median IR is close to 0% (95% CI: [–0.18, 0.71]). The DAG logger is even lighter, with a slightly negative overall median IR (–1.28%, 95% CI: [–2.76, –0.26]), meaning that logged runs often finish as fast or even marginally faster than the baseline due to solver nondeterminism. In all cases, the 95% confidence intervals remain tightly centered around zero, confirming that proof logging does not add meaningful runtime cost.

More precisely:

- **Hard clauses.** For small instances (Q1), both loggers frequently finish slightly faster than the baseline (median IR: –4.36% for standard, –5.32% for DAG). As the number of hard clauses grows, standard logging stabilizes around small positive overheads (Q3–Q4: up to 3.7%), while DAG logging remains statistically indistinguishable from zero (CIs spanning [–4.37, 0.09]).

TABLE I: Summary for advanced by #HardClauses grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	184	180	-5.32 [-12.58, -0.33]	22.7	1.8×10^{-3}	1.22×10^{3}	1.79×10^{4}	2.97×10^{5}	7.56×10^{7}	1.88×10^{4}	4.49×10^{5}
Q2	114	127	129	-1.24 [-8.30, -0.15]	20.1	2.0×10^{-4}	1.96×10^{4}	947	3.77×10^{7}	4.58×10^{9}	3.94×10^{3}	5.63×10^{8}
Q3	102	350	352	-0.04 [-0.56, 4.22]	64.7	2.0×10^{-3}	1.27×10^{5}	3.29×10^{3}	8.20×10^{9}	7.37×10^{11}	4.20×10^{4}	2.87×10^{9}
Q4	108	634	583	-1.96 [-4.37, 0.09]	158	8.3×10^{-2}	3.53×10^{6}	2.69×10^{5}	1.80×10^{8}	8.98×10^{9}	1.56×10^{6}	1.05×10^{7}

TABLE II: Summary for standard by #HardClauses grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	184	188	-4.36 [-7.72, -0.16]	1.15	2.9×10^{-3}	1.22×10^{3}	1.79×10^{4}	2.97×10^{5}	7.56×10^{7}	1.88×10^{4}	4.49×10^{5}
Q2	114	127	136	-0.18 [-2.52, 0.02]	4.63	7.7×10^{-3}	1.96×10^{4}	947	3.77×10^{7}	4.58×10^{9}	3.94×10^{3}	5.63×10^{8}
Q3	102	350	368	1.30 [0.10, 4.49]	7.19	6.5×10^{-3}	1.27×10^{5}	3.29×10^{3}	8.20×10^{9}	7.37×10^{11}	4.20×10^{4}	2.87×10^{9}
Q4	108	634	657	3.67 [0.64, 6.97]	37.0	2.43×10^{-1}	3.53×10^{6}	2.69×10^{5}	1.80×10^{8}	8.98×10^{9}	1.56×10^{6}	1.05×10^{7}

TABLE III: Summary for advanced by #SoftClauses grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	319	322	-0.41 [-2.33, -0.15]	90.9	8.7×10^{-3}	6.89×10^{5}	51.2	3.76×10^{7}	7.58×10^{8}	1.34×10^{5}	1.38×10^{6}
Q2	108	236	233	-0.65 [-4.12, 0.49]	62.0	2.0×10^{-3}	1.54×10^{5}	295	2.87×10^{8}	1.66×10^{10}	4.02×10^{4}	2.50×10^{9}
Q3	108	269	259	-2.58 [-5.44, 0.32]	44.6	5.0×10^{-3}	3.34×10^{5}	1.54×10^{3}	7.64×10^{9}	6.89×10^{11}	2.80×10^{4}	7.65×10^{8}
Q4	108	458	419	-2.66 [-7.78, 0.09]	65.9	7.2×10^{-2}	2.50×10^{6}	2.90×10^{5}	2.00×10^{6}	3.29×10^{9}	1.42×10^{6}	5.06×10^{7}

TABLE IV: Summary for standard by #SoftClauses grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	319	339	0.10 [-0.14, 0.93]	26.0	3.7×10^{-2}	6.89×10^{5}	51.2	3.76×10^{7}	7.58×10^{8}	1.34×10^{5}	1.38×10^{6}
Q2	108	236	264	0.30 [-1.60, 2.78]	4.99	7.5×10^{-3}	1.54×10^{5}	295	2.87×10^{8}	1.66×10^{10}	4.02×10^{4}	2.50×10^{9}
Q3	108	269	274	-1.27 [-4.70, 1.79]	4.66	1.5×10^{-2}	3.34×10^{5}	1.54×10^{3}	7.64×10^{9}	6.89×10^{11}	2.80×10^{4}	7.65×10^{8}
Q4	108	458	459	0.39 [-0.37, 3.31]	14.2	2.0×10^{-1}	2.50×10^{6}	2.90×10^{5}	2.00×10^{6}	3.29×10^{9}	1.42×10^{6}	5.06×10^{7}

- **Soft clauses.** Overhead stays close to zero across all quartiles. Median IRs lie between –2.6% and $+0.4\%$ for both modes, and all confidence intervals overlap with zero, indicating no consistent penalty from logging.
- **Variables.** Again, overhead is negligible. Standard logging has medians in the range of –4.1% to $+0.7\%$, while DAG logging stays between –8.7% and –0.06%. In both cases, 95% CIs show that runtime differences are within solver noise.

In summary, embedding proof logging, either as a human-readable stream or as a compact DAG, adds only a few percent (or effectively zero) to the solving time, confirming that our approach scales seamlessly to competition-level MaxSAT instances.

RQ2: Certificate Size

Figure II–VI and I–V show that the standard proof format produces files in the order of a few megabytes up to 0.2 GB, depending on problem scale. For instance, when grouping by hard-clause quartiles, the median proof size is only 0.003 GB. Even in the largest soft-clause and variable-count bins, it remains under 0.087 GB. In contrast, the advanced format exploits shared derivations to achieve significantly better compaction. Thus, in every grouping its median proof size lies below 0.005 GB, and the 90th-percentile never exceeds 0.025 GB.

When we compare against the certified-CGSS solver (Tbl. VII), this reduction becomes even more noticeable. The PSD column reports that our standard proofs consume roughly 30% of the disk space of the pseudo-Boolean certificates emitted by certified-CGSS (median), while our DAG proofs further shrink to only 3% of the CGSS proof size. Even at the 90th percentile, standard logs are under $6\times$ smaller, and DAG logs remain $10\times$ to $20\times$ smaller.

Overall, the binary-DAG format consistently delivers a 4–$10\times$ reduction over the standard proof logs, and up to $30\times$ smaller certificates than the prior VeriPB-based approach.

RQ3: Verification Cost

Tables II–VI and I–V report absolute verification times for both logging modes across the same quartile groupings. The standard format exhibits fast verification as median times range from about 1 second in the smallest hard-clause bin (Q1) up to 37 s in the largest (Q4), and remain below 15 seconds in all soft-clause and variable-count bins. Even the 90th-percentile standard verifications complete in under one minute on competition-level inputs.

By contrast, the DAG format incurs higher but still manageable checking cost. Median verification times grow from roughly 20 seconds (hard-clauses Q2) to 158 seconds (hard-

TABLE V: Summary for advanced by #Variables grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	45.5	49.2	-8.75 [-14.41, -0.73]	24.4	1.0×10^{-4}	1.09×10^{4}	299	4.31×10^{4}	7.34×10^{5}	662	2.74×10^{5}
Q2	108	232	234	-0.39 [-2.86, -0.07]	53.1	4.3×10^{-3}	3.53×10^{5}	1591	7.77×10^{6}	6.19×10^{8}	3.58×10^{3}	3.08×10^{7}
Q3	114	453	439	-0.06 [-0.91, 0.60]	116	5.8×10^{-3}	3.64×10^{5}	4.35×10^{3}	7.34×10^{9}	6.63×10^{11}	1.61×10^{4}	3.10×10^{9}
Q4	102	556	514	-3.46 [-7.89, -0.37]	67.3	8.1×10^{-2}	3.10×10^{6}	3.02×10^{5}	2.31×10^{8}	1.04×10^{10}	1.69×10^{6}	1.20×10^{7}

TABLE VI: Summary for standard by #Variables grouped in quartiles (IR reported as median with 95% bootstrap confidence interval).

qbin	FC	R(s)	TR(s)	IR_{med} [95% CI]	VR	PS	#HC	#SC	MW	SW	V	OC
Q1	108	45.5	55.0	-4.11 [-10.00, -0.15]	0.30	4.0×10^{-4}	1.09×10^{4}	299	4.31×10^{4}	7.34×10^{5}	662	2.74×10^{5}
Q2	108	232	252	0.03 [-1.09, 1.33]	7.48	2.1×10^{-2}	3.53×10^{5}	1591	7.77×10^{6}	6.19×10^{8}	3.58×10^{3}	3.08×10^{7}
Q3	114	453	471	0.69 [0.07, 2.78]	6.35	1.7×10^{-2}	3.64×10^{5}	4.35×10^{3}	7.34×10^{9}	6.63×10^{11}	1.61×10^{4}	3.10×10^{9}
Q4	102	556	563	0.56 [-1.50, 3.98]	37.5	2.3×10^{-1}	3.10×10^{6}	3.02×10^{5}	2.31×10^{8}	1.04×10^{10}	1.69×10^{6}	1.20×10^{7}

TABLE VII: Comparison against Certified-CGSS

Mode	VO_EMS_{med}	$VO\_EMS_{90\%}$	VO_CGSS_{med}	$VO\_CGSS_{90\%}$	VSP_{med}	$VSP_{90\%}$	PSD_{med}	$PSD_{90\%}$
normal	0.01	0.98	5.64	42.20	0.02	1.63	0.30	5.78
dag	0.59	1.04	5.64	42.20	0.16	14.04	0.03	1.0

clauses Q4), and span 65 to 91 seconds in the largest soft-clause bins, and 24 to 67 seconds in variable-count bins. All DAG proofs validate within three minutes.

When compared against the certified-CGSS baseline in Table VII, the difference is stands out. Our standard proofs exhibit a median slowdown (VO_EMS) of only $0.01\times$ solve time (90th: $0.98\times$), while certified-CGSS's pseudo-Boolean certificates incur $5.64\times$ (90th: $42.20\times$). Even the DAG mode, at $0.59\times$ (90th: $1.04\times$), verifies over ten times faster than certified-CGSS. The element-wise ratio VSP shows that our verifier runs dramatically faster than certified-CGSS: the standard checker achieves a median speedup of over $587\times$ (90th percentile: 3.2×10^4), while the DAG checker reaches a median of $20\times$ (90th percentile: 6.3×10^3).

In summary, both proof formats enable rapid, end-to-end certification. the standard logger offers near-instant validation, and the DAG log, while incurring higher cost, remains an order of magnitude quicker than the existing VeriPB-based method, solidifying our framework's suitability for large-scale MaxSAT certification.

E. Limitations and Outlook

Our rules currently target the empirically dominant OLL family of MaxSAT solvers. Certification for other paradigms (e.g., implicit hitting set, branch-and-bound) exists and/or is possible but remains future work.

Our checker replays SCIP calls as black boxes on recorded intermediate instances. We do not trust the original solver calls but perform fresh ones with the recorded assumptions to verify the result. For SCIP, no standard proof format exists, and defining one is an important direction for future research.

Finally, while our evaluation covers the full Weighted MaxSAT 2024 track, industrial instances may differ structurally. Additional experiments on domain-specific benchmarks (e.g., software verification, testing, or configuration) need to be valuable to confirm the practical impact of our framework.

VII. Conclusion

We have introduced the first MaxSAT-specific proof-logging framework tailored to core-guided OLL solving, supporting both a human-readable format and a compact binary dependency DAG. Our system captures every inference rule, enabling lightweight proof generation and dramatically faster verification, with speedups of up to three orders of magnitude compared to prior pseudo-Boolean approaches based on VeriPB. On the full 2024 MaxSAT Competition suite, both logging modes successfully verified every solved instance, with normal proofs incurring median verification cost of only 1% of solve time and DAG proofs reducing certificate size by up to three orders of magnitude at the 90th percentile.

Beyond competition benchmarks, our contributions directly support software engineering workflows that increasingly rely on MaxSAT, such as predicate abstraction in program verification, test-suite minimization, and automated debugging. The human-readable logs provide transparency for developers and auditors, while the DAG format offers scalable certificates suitable for continuous integration pipelines, regulatory compliance, and bug analysis. By closing the gap between high-performance MaxSAT solving and rigorous certification, our framework enables trustworthy optimization in safety-critical and auditable contexts, making MaxSAT a more reliable foundation for modern software engineering tasks.

VIII. Acknowledgments

We thank ASE reviewers for their insightful comments. The work of Ruzica Piskac and Timos Antonopoulos was partially supported by the National Science Foundation under Grant Numbers CCF-2318974, CCF-2219995, and CNS-2245344.

REFERENCES

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Logic, automata, and computational complexity: The works of Stephen A. Cook*, 2023, pp. 143–152.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 1999, pp. 193–207.

[3] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta, "Combining decision diagrams and sat procedures for efficient symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 124–138.

[4] F. Arito, F. Chicano, and E. Alba, "On the application of sat solvers to the test suite minimization problem," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 45–59.

[5] A. Yamada, T. Kitamura, C. Artho, E.-H. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental sat solving," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[6] S. Narain *et al.*, "Network configuration management via model finding." in *LISA*, vol. 5, 2005, pp. 15–15.

[7] S. Yatkın and T. Ovatman, "Logical analysis and contradiction detection in high-level requirements during the review process using sat-solver," *arXiv preprint arXiv:2405.00163*, 2024.

[8] M. J. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2013.

[9] ——, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, 2013.

[10] V. D'Silva, L. Haller, and D. Kroening, "Satisfiability solvers are static analysers," in *International Static Analysis Symposium*. Springer, 2012, pp. 317–333.

[11] P. C. Attie, K. D. A. Bab, and M. Sakr, "Model and program repair via sat solving," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–25, 2017.

[12] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 173–188.

[13] M. J. Heule, M. Iser, M. Järvisalo, and M. Suda, "Proceedings of sat competition 2024: Solver, benchmark and proof checker descriptions," 2024.

[14] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for cnf formulas," in *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2003, pp. 886–891.

[15] M. Järvisalo, M. J. Heule, and A. Biere, "Inprocessing rules," in *International Joint Conference on Automated Reasoning*. Springer, 2012, pp. 355–370.

[16] M. J. Heule, W. A. Hunt Jr, and N. Wetzler, "Verifying refutations with extended resolution," in *International Conference on Automated Deduction*. Springer, 2013, pp. 345–359.

[17] N. Wetzler, M. J. Heule, and W. A. Hunt Jr, "Drat-trim: Efficient checking and trimming using expressive clausal proofs," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 422–429.

[18] M. J. Heule, W. A. Hunt Jr, and N. Wetzler, "Verifying refutations with extended resolution," in *International Conference on Automated Deduction*. Springer, 2013, pp. 345–359.

[19] M. J. Heule, W. A. Hunt, and N. Wetzler, "Trimming while checking clausal proofs," in *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 181–188.

[20] S. C. Committee, "General rules," 2021, retrieved May 26, 2025 from https://satcompetition.github.io/2021/rules.html.

[21] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 437–446, 2011.

[22] E. Zhai, R. Piskac, R. Gu, X. Lao, and X. Wang, "An auditing language for preventing correlated failures in the cloud," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–28, 2017.

[23] K. Yin, H. Zhang, X. Fang, Y. Shi, T. S. Humble, A. Li, and Y. Ding, "Qecc-synth: A layout synthesizer for quantum error correction codes on sparse architectures," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 876–890.

[24] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[25] C. M. Li and F. Manya, "Maxsat, hard and soft constraints," in *Handbook of satisfiability*. IOS Press, 2021, pp. 903–927.

[26] J. Berg, M. Järvisalo, R. Martins, A. Niskanen, and T. Paxian, "Maxsat evaluation 2024: Solver and benchmark descriptions," 2024.

[27] H. Ihalainen, A. Oertel, Y. K. Tan, J. Berg, M. Järvisalo, M. O. Myreen, and J. Nordström, "Certified maxsat preprocessing," in *International Joint Conference on Automated Reasoning*. Springer, 2024, pp. 396–418.

[28] E. Boros and P. L. Hammer, "Pseudo-boolean optimization," *Discrete applied mathematics*, vol. 123, no. 1-3, pp. 155–225, 2002.

[29] B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström, "Certified dominance and symmetry breaking for combinatorial optimisation," *Journal of Artificial Intelligence Research*, vol. 77, pp. 1539–1589, Aug. 2023, preliminary version in *AAAI '22*.

[30] S. Gocht and J. Nordström, "Certifying parity reasoning efficiently using pseudo-Boolean proofs," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, Feb. 2021, pp. 3768–3777.

[31] S. Gocht, "Certifying correctness for combinatorial algorithms by using pseudo-Boolean reasoning," Ph.D. dissertation, Lund University, Lund, Sweden, Jun. 2022, available at https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu.

[32] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided maxsat with soft cardinality constraints," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2014, pp. 564–573.

[33] X. Si, X. Zhang, R. Grigore, and M. Naik, "Maximum satisfiability in software analysis: Applications and techniques," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 68–94.

[34] R. Martins, S. Joshi, V. Manquinho, and I. Lynce, "Reflections on" incremental cardinality constraints for maxsat"," *arXiv preprint arXiv:1910.04643*, 2019.

[35] O. Bailleux and Y. Boufkhad, "Efficient cnf encoding of boolean cardinality constraints," in *International conference on principles and practice of constraint programming*. Springer, 2003, pp. 108–122.

[36] R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer, "Certifying algorithms," *Computer Science Review*, vol. 5, no. 2, pp. 119–161, 2011.

[37] E. Alkassar, S. Böhme, K. Mehlhorn, C. Rizkallah, and P. Schweitzer, "An introduction to certifying algorithms," *it-Information Technology*, vol. 53, no. 6, pp. 287–293, 2011.

[38] F. Avellaneda, "A short description of the solver evalmaxsat," *MaxSAT Evaluation*, vol. 8, p. 364, 2020.

[39] M. Piotrów, "Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2020, pp. 132–136.

[40] Z. Lei, S. Cai, D. Wang, Y. Peng, F. Geng, D. Wan, Y. Deng, and P. Lu, "Cashwmaxsat: Solver description," *MaxSAT Evaluation*, vol. 2021, p. 8, 2021.

[41] C.-M. Li, Z. Xu, J. Coll, F. Manyà, D. Habet, and K. He, "Boosting branch-and-bound maxsat solvers with clause learning," *AI Communications*, vol. 35, no. 2, pp. 131–151, 2022.

[42] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into sat," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 2, no. 1-4, pp. 1–26, 2006.

[43] D. Le Berre and A. Parrain, "The sat4j library, release 2.2: System description," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2011.

[44] T. Paxian, S. Reimer, and B. Becker, "Dynamic polynomial watchdog encoding for solving weighted maxsat," in *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 21*. Springer, 2018, pp. 37–53.

[45] J. Davies and F. Bacchus, "Exploiting the power of mip solvers in maxsat," in *Theory and Applications of Satisfiability Testing–SAT 2013: 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings 16*. Springer, 2013, pp. 166–181.

[46] ——, "Postponing optimization to speed up maxsat solving," in *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings 19*. Springer, 2013, pp. 247–262.
[47] M. Alviano, C. Dodaro, F. Ricca *et al.*, "A maxsat algorithm using cardinality constraints of bounded size." in *IJCAI*, vol. 2015, 2015, pp. 2677–2683.
[48] C. Ansótegui and J. Gabàs, "Wpm3: an (in) complete algorithm for weighted partial maxsat," *Artificial Intelligence*, vol. 250, pp. 37–57, 2017.
[49] Z. Fu and S. Malik, "On solving the partial max-sat problem," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2006, pp. 252–265.
[50] N. Narodytska and F. Bacchus, "Maximum satisfiability using core-guided maxsat resolution," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, 2014.
[51] M. L. Bonet, J. Levy, and F. Manyà, "Resolution for max-sat," *Artificial Intelligence*, vol. 171, no. 8-9, pp. 606–618, 2007.
[52] Y. Filmus, M. Mahajan, G. Sood, and M. Vinyals, "Maxsat resolution and subcube sums," *ACM Transactions on Computational Logic*, vol. 24, no. 1, pp. 1–27, 2023.
[53] H. Ihalainen, J. Berg, and M. Järvisalo, "Clause redundancy and preprocessing in maximum satisfiability," in *International Joint Conference on Automated Reasoning*. Springer, 2022, pp. 75–94.
[54] M. Py, M. S. Cherif, and D. Habet, "Proofs and certificates for max-sat," *Journal of Artificial Intelligence Research*, vol. 75, pp. 1373–1400, 2022.
[55] ——, "A proof builder for max-sat," in *Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings 24*. Springer, 2021, pp. 488–498.
[56] ——, "Towards bridging the gap between sat and max-sat refutations," in *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2020, pp. 137–144.
[57] A. Morgado and J. Marques-Silva, "On validating boolean optimizers," in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*. IEEE, 2011, pp. 924–926.
[58] A. Morgado, A. Ignatiev, M. L. Bonet, J. Marques-Silva, and S. Buss, "Drmaxsat with maxhs: first contact," in *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*. Springer, 2019, pp. 239–249.
[59] J. Larrosa, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "A framework for certified boolean branch-and-bound optimization," *Journal of Automated Reasoning*, vol. 46, no. 1, pp. 81–102, 2011.
[60] J. Berg, B. Bogaerts, J. Nordström, A. Oertel, T. Paxian, and D. Vandesande, "Certifying without loss of generality reasoning in solution-improving maximum satisfiability," in *International Conference on Principles and Practice of Constraint Programming*. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2024, p. 4.
[61] D. Vandesande, W. De Wulf, and B. Bogaerts, "Qmaxsatpb: A certified maxsat solver," in *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2022, pp. 429–442.
[62] H. Ihalainen, J. Berg, and M. Järvisalo, "Refined core relaxation for core-guided maxsat solving," in *CP*, ser. LIPIcs, vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 28:1–28:19.
[63] J. Berg, B. Bogaerts, J. Nordström, A. Oertel, and D. Vandesande, "Certified core-guided maxsat solving," in *International Conference on Automated Deduction*. Springer, 2023, pp. 1–22.
[64] D. Vandesande and B. Bogaerts, "Towards certified maxsat solving," 2023.