Certified Branch-and-Bound MaxSAT Solving

Dieter Vandesande[1,2], **Jordi Coll**[3], **Bart Bogaerts**[2,1]

[1]Vrije Universiteit Brussel, Dept. of Computer Science, Brussels, Belgium
[2]KU Leuven, Dept. of Computer Science, B-3000 Leuven, Belgium
[3]Universitat de Girona, Girona, Spain
dieter.vandesande@vub.be, jordicoll@udg.edu, bart.bogaerts@kuleuven.be

Abstract

Over the past few decades, combinatorial solvers have seen remarkable performance improvements, enabling their practical use in real-world applications. In some of these applications, ensuring the correctness of the solver's output is critical. However, the complexity of modern solvers makes them susceptible to bugs in their source code. In the domain of satisfiability checking (SAT), this issue has been addressed through *proof logging*, where the solver generates a formal proof of the correctness of its answer. For more expressive problems like MaxSAT, the optimization variant of SAT, proof logging had not seen a comparable breakthrough until recently.

In this paper, we show how to achieve proof logging for state-of-the-art techniques in Branch-and-Bound MaxSAT solving. This includes certifying look-ahead methods used in such algorithms as well as advanced clausal encodings of pseudo-Boolean constraints based on so-called Multi-Valued Decision Diagrams (MDDs). We implement these ideas in MaxCDCL, the dominant branch-and-bound solver, and experimentally demonstrate that proof logging is feasible with limited overhead, while proof checking remains a challenge.

1 Introduction

With increasingly efficient solvers being developed across various domains of combinatorial search and optimization, we have effectively reached a point where NP-hard problems are routinely addressed in practice. This maturation has led to the deployment of solvers in a wide range of real-world applications, including safety-critical systems and life-impacting decision-making processes—such as verifying software for transportation infrastructure (Ferreira and Silva 2004), checking the correctness of plans for the space shuttle's reaction control system (Nogueira et al. 2001), and matching donors with recipients in kidney transplants (Manlove and O'Malley 2014). Given these high-stakes applications, it is crucial that the results produced by such solvers are guaranteed to be correct. Unfortunately, this is not always the case: numerous reports have documented solvers producing infeasible solutions or incorrectly claiming optimality or unsatisfiability (Berg et al. 2023; Brummayer, Lonsing, and Biere 2010; Cook et al. 2013; Gillard, Schaus, and Deville 2019; Järvisalo, Heule, and Biere 2012).

This problem calls for a systematic solution. The most straightforward approach would be to use *formal verification*; that is, employing a proof assistant (Bertot and Castéran 2004; Nipkow, Paulson, and Wenzel 2002; Slind and Norrish 2008; de Moura and Ullrich 2021) to formally prove the correctness of a solver. However, in practice, formal verification often comes at the cost of performance (Fleury 2020; Heule et al. 2022), which is precisely what has driven the success of combinatorial optimization.

Instead, we advocate for the use of *certifying algorithms* (McConnell et al. 2011), an approach also known as *proof logging* in the context of combinatorial optimization. With proof logging, a solver not only produces an answer (e.g., an optimal solution to an optimization problem) but also a *proof of correctness* for that answer. This proof can be checked efficiently (in terms of the proof size) by an independent tool known as a *proof checker*. Beyond ensuring correctness, proof logging also serves as a powerful software development methodology: it supports advanced *testing* and *debugging* of solver implementations (Brummayer and Biere 2009). Moreover, the generated proofs constitute an auditable trail and can be used to speed-up the generation of explanations of why a particular conclusion was reached (Bleukx et al. 2026).

Proof logging was pioneered in the field of Boolean satisfiability (SAT), where numerous proof formats and proof checkers, including formally verified ones, have been developed over the years (Biere 2006; Goldberg and Novikov 2003; Heule, Hunt, and Wetzler 2013; Wetzler, Heule, and Hunt 2014; Cruz-Filipe et al. 2017). For several years, proof logging has been mandatory in the main tracks of the annual SAT solving competition, reflecting the community's strong commitment to ensuring that all SAT solvers are certifying.

In this paper, we are concerned with *maximum satisfiability* (MaxSAT), the optimization variant of SAT. In contrast to SAT, proof logging in MaxSAT is still relatively uncommon. While several proof systems for MaxSAT have been developed (Bonet, Levy, and Manyà 2007; Larrosa et al. 2011; Morgado and Marques-Silva 2011; Py, Cherif, and Habet 2020) and a solver has been designed specifically to generate such proofs (Py, Cherif, and Habet 2022), only with the recent advent of the VERIPB proof system (Bogaerts et al. 2023; Gocht and Nordström 2021) did a general-purpose proof logging methodology for MaxSAT see the light (Van-

desande, De Wulf, and Bogaerts 2022). VERIPB, a proof system for linear inequalities over 0–1 integer variables, has since been successfully applied to MaxSAT solvers based on *solution-improving search* (Vandesande, De Wulf, and Bogaerts 2022; Vandesande 2023; Berg et al. 2024a) (where a SAT oracle is repeatedly queried to find solutions that improve upon the current best), as well as to solvers using *core-guided search* (Berg et al. 2023) (where a SAT oracle is queried iteratively under increasingly relaxed optimistic assumptions). The techniques developed there have also led to certification of generalizations of MaxSAT for multi-objective problems (Jabs et al. 2025).

In addition to solution-improving and ore-guided search, state-of-the-art MaxSAT solvers also employ other strategies, such as *implicit hitting set* and *branch-and-bound* search. In a companion paper (Ihalainen et al. 2026), we show how to certify IHS search, and in the current paper, we develop VERIPB-based certification for branch-and-bound search (Li and Manyà 2021; Li et al. 2022; Abramé and Habet 2014; Kügel 2010; Li, Manyà, and Planes 2007; Heras, Larrosa, and Oliveras 2008), thereby finally covering all key search paradigm in MaxSAT solving. Modern branch-and-bound solvers combine conflict-driven clause learning (CDCL) (Silva and Sakallah 1999) with a sophisticated bounding function that determines whether it is the case that no assignment refining the current node can improve upon the best solution found so far (Li et al. 2022, 2021; Coll et al. 2025b). While it is well understood how to certify CDCL search using VERIPB, the main challenge lies in certifying the conclusions drawn by the bounding function. This bounding function employs look-ahead methods to generate (conditional) unsatisfiable cores: sets of literals that cannot simultaneously be true. These cores are then combined to estimate the best possible objective value that remains achievable.

In order to bring this certification to practice, there are more hurdles to overcome: state-of-the-art branch-and-bound solvers employ several other clever tricks to speed up the solving process, such as pre-processing methods as well as integrating ideas from other search paradigms. One particular technique that proved to be challenging is the use of *multi-valued decision diagrams* (MDDs) in order to create a CNF encoding of a solution-improving constraint (which is enabled or disabled heuristically depending on the size of the instance at hand) (Bofill et al. 2020). This encoding generalizes the encoding of Abío et al. (2012) based on Binary Decision Diagrams (BDDs) by allowing splits on sets of variables instead of a single variable. While MDDs have been used in a lazy clause generation setting (Gange, Stuckey, and Szymanek 2011), their clausal encoding has never been certified. From the perspective of proof logging, one main challenge with BDD- or MDD-based encodings is that multiple (equivalent) constraints are represented by the same variable in this encoding. In the proof, we then have to show that they are indeed equivalent. As an example, consider the constraints

$$12x_1 + 5x_2 + 4x_3 \geq 6 \qquad \text{and}$$
$$12x_1 + 5x_2 + 4x_3 \geq 9.$$

Taking into account that the variables x_i take values in $\{0, 1\}$, it is not hard to see that these constraints are equivalent: all combinations of truth assignments that lead to the left-hand side taking a value at least six, must assign it a value of at least nine. In other words: the left-hand side cannot take values 6, 7, or 8. In general, checking whether such a linear expression can take a specific value (e.g., 7) is well-known to be NP hard, but BDD (and MDD) construction algorithms will in several cases detect this efficiently. The main question of interest for us is how to convince a proof checker of the fact that these two constraints are equivalent, without doing a substantial amount of additional work. We achieve this using an algorithm that proves this property for all nodes in an MDD in a linear pass over its representation.

To evaluate our approach, we add proof logging to the branch-and-bound solver WMaxCDCL (Coll et al. 2025b). WMaxCDCL is the current state-of-the-art in branch-and-bound MaxSAT solving, which is showcased by the fact that it won (as part of a portfolio-solver) the unweighted track of the 2024 edition and the weighted track of the 2023 edition (Järvisalo et al. 2023; Berg et al. 2024b). In the experiments, we focus on evaluating the overhead in the solving time, while also measuring the time necessary to check the produced proofs; the results show that proof logging is indeed possible without significant overhead for most cases, while proof checking overhead can be improved.

The rest of this paper is structured as follows. In Section 2, we recall some preliminaries about MaxSAT solving and VERIPB-based proof logging. Section 3 is devoted to presenting the core of the MAXCDCL algorithm, focusing on look-ahead–based bounding, as well as explaining how to get a certifying version of this. In Section 4, we explain how BDDs and MDDs are used to encode PB constraints and how this can be made certifying. Section 5 contains our experimental analysis and Section 6 concludes the paper.

Formal details and proofs are often omitted, but can be found in the extended version of this paper (Vandesande, Coll, and Bogaerts 2025a).

2 Preliminaries

We first recall some concepts from pseudo-Boolean optimization and MaxSAT solving. Afterwards, we introduce the VERIPB proof system. A more complete exposition can be found in previous work (Li and Manyà 2021; Bogaerts et al. 2023).

2.1 Pseudo-Boolean Optimization and MaxSAT

In this paper, all variables are assumed to be *Boolean*; meaning they take a value in $\{0, 1\}$. A *literal* ℓ is a Boolean variable x or its negation $\overline{x}$. A *pseudo-Boolean (PB) constraint* C is a 0–1 integer linear inequality $\sum_i w_i \ell_i \bowtie A$, with w_i, A integers, ℓ_i literals and $\bowtie \in \{\geq, \leq\}$. Without loss of generality, we will often assume our constraints to be *normalized*, meaning that the ℓ_i are different literals, the comparison symbol $\bowtie$ is equal to $\geq$ and all coefficients w_i and the degree A are non-negative. A *formula* is a conjunction of PB constraints. A *clause* is a special case of a PB constraint having all w_i and A equal to one.

A *(partial) assignment* α is a (partial) function from the set of variables to $\{0, 1\}$; it is extended to literals in the obvious way, and we sometimes identify an assignment with the set of literals it maps to 1. We write $C|_\alpha$ for the constraint obtained from C by substituting all assigned variables x by their assigned value $\alpha(x)$. A constraint C is *satisfied* under α if $\sum_{\alpha(\ell_i)=1} w_i \geq A$. A formula F is satisfied under α if all of its constraints are. We say that F *implies* C (and write $F \models C$) if all assignments that satisfy F also satisfy C. A constraint C *(unit) propagates* a literal x to a value in $\{0, 1\}$ under a (partial) assignment α if assigning x the opposite value makes the constraint $C|_\alpha$ unsatisfiable.

A *pseudo-Boolean optimization* instance consists of a formula F and a linear term $\mathcal{O} = \sum_i v_i b_i$ (called the *objective*) to be minimized, where v_i are integers (without loss of generality assumed to be *positive* here) and b_i are different literals, which we will refer to as *objective literals*. If ℓ is a literal, we write $w_\mathcal{O}(\ell)$ for the weight of ℓ in $\mathcal{O}$, i.e., $w_\mathcal{O}(b_i) = v_i$ for all i and $w_\mathcal{O}(\ell) = 0$ for all other literals. In line with previous work on certifying MaxSAT solvers (e.g., Berg et al. 2023; Vandesande 2023; Berg et al. 2024a), in this paper, we view MaxSAT as the special case of pseudo-Boolean optimization where F is a conjunction of clauses. A discussion on the equivalence of this view and the more clause-centric view using hard and soft clauses is given by for example Leivo, Berg, and Järvisalo (2020).

2.2 The VERIPB Proof System

For a pseudo-Boolean optimization instance $(F, \mathcal{O})$, the VERIPB proof system (Bogaerts et al. 2023; Gocht and Nordström 2021) maintains a *proof configuration* $\mathcal{F}$, which is a set of constraints derived so far, and is initialized with F. We are allowed to update the configuration using the cutting planes proof system (Cook, Coullard, and Turán 1987):

Literal Axioms: For any literal, we can add $\ell_i \geq 0$ to $\mathcal{F}$.

Linear Combination: Given two PB constraints C_1 and C_2 in $\mathcal{F}$, we can add a positive linear combination of C_1 and C_2 to $\mathcal{F}$.

Division: Given the normalized PB constraint $\sum_i w_i \ell_i \geq A$ in $\mathcal{F}$ and a positive integer c, we can add the constraint $\sum_i \lceil w_i/c \rceil \ell_i \geq \lceil A/c \rceil$ to $\mathcal{F}$.

Saturation: Given the normalized PB constraint $\sum_i a_i l_i \geq A$ in $\mathcal{F}$, we can add $\sum_i b_i l_i \geq A$ with $b_i = \min(a_i, A)$ for each i, to $\mathcal{F}$.

Importantly, in many cases, it is not required to show precisely why a constraint can be derived, but the verifier can figure it out itself by means of so-called **reverse unit propagation (RUP)** (Goldberg and Novikov 2003): if applying unit propagation until fixpoint on $\mathcal{F} \wedge \neg C$ propagates to a contradiction, then we can add C to $\mathcal{F}$.

VERIPB has some additional rules, which are briefly discussed below. We refer to earlier work (Bogaerts et al. 2023) for more details. There is a rule for dealing with optimization statements:

Objective Improvement: Given an assignment α that satisfies $\mathcal{F}$, we can add the constraint $\mathcal{O} < \mathcal{O}|_\alpha$ to $\mathcal{F}$.

The constraint added to $\mathcal{F}$ is also called the solution-improving constraint; if $0 \geq 1$ can be derived from $\mathcal{F}$ after applying the objective improvement rule, then it is concluded that α is an optimal solution.

VERIPB allows deriving non-implied constraints with the *redundance-based strengthening rule*: a generalization of the RAT rule (Järvisalo, Heule, and Biere 2012) and commonly used in proof systems for SAT. In this paper, the redundance rule is mainly used for **reification**: for any PB constraint C and any fresh variable v, not used before, two applications of the redundance rule allow us to derive PB constraints that express $v \Rightarrow C$ and $v \Leftarrow C$. Moreover, the redundance rule allows us to derive implied constraints by a **proof by contradiction**: if we have a cutting planes derivation that shows that $\mathcal{F} \wedge \neg C \models 0 \geq 1$, then we can add C to $\mathcal{F}$.[1]

3 Certifying Branch-And-Bound with Look-Ahead

In this section, we present the working of modern branch-and-bound search for MaxSAT, specifically for the MAXCDCL algorithm. Our presentation is given with proof logging in mind (we immediately explain what is essential for each step from the proof logging perspective) and hence we often take a different perspective from the original papers introducing this algorithm (Li et al. 2021; Coll et al. 2025b).

MAXCDCL combines branch-and-bound search and conflict-driven clause learning (CDCL). An overview of the algorithm is given in Algorithm 1. As can be seen, this essentially performs CDCL search (this is the *branching* aspect), using standard techniques such as assigning variables, unit propagation, conflict analysis, and non-chronological backtracking (Silva and Sakallah 1999).

MAXCDCL maintains the objective value v^* of the best found solution so far, which is initialized as $+\infty$. However, after unit propagation the `lookahead` procedure is called. This procedure is where the *bounding* aspect happens: here sophisticated techniques are used to determine whether the current assignment can still be extended to a satisfying assignment that improves upon the current best objective value. If not, the search is interrupted and a new clause forcing the solver to backtrack is learned.

It is well known how to get VERIPB-based certification for CDCL (this is done for instance in a MaxSAT setting by Vandesande, De Wulf, and Bogaerts 2022), so we focus here only on how to get proof logging for the `lookahead` method.[2]

Let us now focus on the `lookahead` procedure. Before diving into the algorithmic aspects, we formalize what it aims to compute, namely a set of *weighted local cores*.

Definition 1. *Let α be an assignment, and $(F, \mathcal{O})$ a MaxSAT instance. A* weighted local core *of $(F, \mathcal{O})$ relative to α is a*

[1] In practice, this is mimicked by applying the *redundance-based strengthening rule* with an empty witness; details can be found in the supplementary material.

[2] Obviously, our implementation with proof logging also handles the other aspects of the algorithm.

Input: CNF formula F, objective $\mathcal{O}$
Output: optimal assigment or **UNSAT**

```
1  F ← Preprocess (F);
2  α ← ∅;   α* ← UNSAT;   v* ← +∞;
3  while true do
4  |  unit-propagate (F, α);
5  |  lookahead (F, O, v*, α);
6  |  if conflict detected then
7  |  |  if at root level then
8  |  |  |  return α*
9  |  |  Analyze conflict and backtrack;
10 |  else if all variables assigned then
11 |  |  if O|_α < v* then α* ← α;   v* ← O|_α;
12 |  |  Backtrack
13 |  else
14 |  |  Decide on some unassigned variable;
```

Algorithm 1: Overview of the MaxCDCL algorithm for branch-and-bound MaxSAT solving.

triple

$$q = \langle w, R, K \rangle$$

such that $R \subseteq \alpha$ *and* K *contains only* negations of *objective literals and*

$$F \wedge R \wedge K \models \bot,$$

where $\bot$ *denotes the trivially false constraint* $0 \geq 1$.

In other words, a local core guarantees that, given the assignments in R (which are also called the reasons of K), the objective literals in K cannot all be false, and hence at least one of the objective literals in K has to incur cost.

Now, we are not just interested in finding a single local core, but in a compatible *set* of such cores.

Definition 2. *Let* $\mathcal{Q}$ *be a set of weighted local cores of* $(F, \mathcal{O})$ *relative to* α. *We call* $\mathcal{Q}$ $\mathcal{O}$*-compatible, if for each objective literal* ℓ,

$$\sum_{\langle w,R,K\rangle \in \mathcal{Q} \wedge \overline{\ell} \in K} w \leq w_{\mathcal{O}}(\ell).$$

In other words, $\mathcal{Q}$ is $\mathcal{O}$-compatible if the total weight of all cores containing a literal does not exceed the weight of the underlying objective literal in the objective. In what follows, if ℓ is an objective literal, we write $res(\ell, \mathcal{Q})$ for

$$w_{\mathcal{O}}(\ell) - \sum_{\langle w,R,K\rangle \in \mathcal{Q} \wedge \overline{\ell} \in K} w$$

and call this the *residual* weight of ℓ with respect to $\mathcal{Q}$. The total *weight* of a core set, written $w_{\mathcal{O}}(\mathcal{Q})$ is $\sum_{\langle w,R,K\rangle \in \mathcal{Q}} w$. The following proposition now formalizes how a set of weighted local cores can be used during search.

Proposition 3. *Let* $\mathcal{Q}$ *be an* $\mathcal{O}$*-compatible set of weighted local cores and assume* β *is any total assignments that refines* α *and satisfies* F.

1. *If* $w_{\mathcal{O}}(\mathcal{Q}) \geq v^*$, *then* $\mathcal{O}|_\beta \geq v^*$.
2. *If for some unassigned objective literal* ℓ,

$$res(\ell, \mathcal{Q}) + w_{\mathcal{O}}(\mathcal{Q}) \geq v^*,$$

then it holds that if $\mathcal{O}|_\beta < v^*$, *then* $\beta(\ell) = 0$.

The first case of this proposition is known as a *soft conflict* (Coll et al. 2025a). This means that if the total weight of the core set exceeds v^*, then we are sure that all assignments that refine the current one will have an objective value that does not improve upon the best found so far. In this case, we know that the current node of the search tree is hopeless and we can backtrack (in fact, MaxCDCL will learn a clause explaining this soft conflict, as discussed later). The second case in Proposition 3 was called *hardening* (Coll et al. 2025a). The setting captured here is that *if* we were to set a literal ℓ to be cost-incurring, then the condition in the first item would trigger. As such, we can safely propagate ℓ to be non-cost-incurring.

Pseudo-code of the lookahead procedure, and details as to how it finds those local cores can be found in Algorithm 2. It initializes a local assignment λ to be equal to α, and maintains a partially constructed core K, initialized as the empty set. In line 4 it initializes $\mathcal{Q}$ to be the set of all trivial weighted local cores (by going over all objective literals that are already cost-incurring in α). In the main loop at line 5, it iteratively assigns an objective literal to be non cost-incurring and runs unit propagation. If this results in another objective literal to be propagated to be cost-incurring (line 8) or if a conflict is found (line 10) , this means we have found a local core. In principle taking R equal to α and K as constructed so far could be used for a local core. Instead, however, line 12 uses standard conflict analysis methods to further reduce this. The local core that is found in this way is added to $\mathcal{Q}$ (line 14) and K and λ are reset (line 16).

If one of the two conditions of Theorem 3 is satisfied, a new clause is learned (based on the set of local cores and standard conflict analysis techniques) and added to F (lines 17–21). Here, we use the notation

$$C_{\mathcal{Q}} := \bigvee_{\langle w,R,K\rangle \in \mathcal{Q} \wedge \ell \in R} \overline{\ell}.$$

We now turn our attention to proof logging for this lookahead procedure. It consists of two important components. On the one hand, we need proofs for the *core discovery* (lines 3–16) and on the other hand, the proof that the clause added to F can indeed be derived. We now show that this can indeed be done efficiently.

Proposition 4. *If* $\langle w, R, K \rangle$ *is a local core found by Algorithm 2, then the clause*

$$C_q := \bigvee_{\ell \in R \cup K} \overline{\ell}.$$

is RUP wrt F.

Proof sketch. This follows from the fact that Algorithm 2 finds a local core q after the propagation of the negation of C_q leads to a contradiction. □

Theorem 5. *Let* $\mathcal{Q}$ *be an* $\mathcal{O}$*-compatible set of weighted local cores and let* $|\mathcal{O}|$ *be the number of objective literals.*

1. *If* $w_{\mathcal{O}}(\mathcal{Q}) \geq v^*$, *then there is a cutting planes derivation that derives* $C_{\mathcal{Q}}$ *from the constraint* $\mathcal{O} \leq v^* - 1$ *and the constraints* C_q *for every* $q \in \mathcal{Q}$ *using at most* $3|\mathcal{O}| + 2|\mathcal{Q}| + 1$ *steps.*

Input: Formula F, objective $\mathcal{O}$, upper bound v^*, assignment α

```
1  λ ← α;   Q ← ∅;
2  K ← ∅;
3  for each objective literal ℓ s.t. α(ℓ) = 1 do
4  |  Q ← Q ∪ {⟨w_O(ℓ), {ℓ}, {ℓ̄}⟩}
5  while some ℓ with res(ℓ, Q) > 0 is unassigned do
6  |  λ(ℓ) ← 0;   K ← K ∪ {ℓ̄};
7  |  unit-propagate (F, λ);
8  |  if some ℓ' with res(ℓ', Q) > 0 is propagated then
9  |  |  K ← K ∪ {ℓ̄'};
10 |  else if no conflict was derived then
11 |  |  continue
12 |  R, K ← improve-core(K);
13 |  w ← min_{ℓ̄∈K} res(ℓ, Q);
14 |  Q ← Q ∪ {⟨w, R, K⟩};
15 |  if w_O(Q) ≥ v* then break;
16 |  K ← ∅;   λ ← α
17 if w_O(Q) ≥ v* then
18 |  F ← F ∪ {analyze(C_Q)};
19 else
20 |  for each ℓ with res(ℓ, Q) + w_O(Q) ≥ v* do
21 |  |  F ← F ∪ {analyze(C_Q ∨ ℓ̄)} ;
22 |  |  unit-propagate (F, α);
```

Algorithm 2: Overview of the `lookahead` procedure.

2. *If for some unassigned objective literal ℓ, $res(\ell, \mathcal{Q}) + w_{\mathcal{O}}(\mathcal{Q}) \geq v^*$, then there is a cutting planes derivation that derives $C_{\mathcal{Q}} \vee \overline{\ell}$ from the constraint $\mathcal{O} \leq v^* - 1$ and the constraints C_q for every $q \in \mathcal{Q}$ using at most $3|\mathcal{O}| + 2|\mathcal{Q}| - 2$ steps.*

Note that since the clauses that are finally learned by MAXCDCL in lines 18 and 21 of Algorithm 2 are derived by standard CDCL on the clauses derived in Theorem 5, they can be proven by RUP. Moreover, when line 8 of Algorithm 1 finds a conflict at root level, the clause $0 \geq 1$ is also provable by RUP, which proves the optimality of the last found solution due to the objective improvement rule.

We end this section with an example illustrating the cutting planes derivation of the first case of Theorem 5.[3]

Example 6. *Let us consider a MaxSAT instance expressed as a PBO instance $(F, \mathcal{O})$, with $\mathcal{O} = 3x_1 + 5x_2 + 5x_3 + 6x_4$. Assume that the best found solution so far has objective value $v^* = 7$, that the current partial assignment assigns y_1 and y_2 to true, and that algorithm 2 has found a set of local cores $\mathcal{Q} = \{q_1, q_2\}$, with*

$$q_1 = \langle 3, \{y_1\}, \{\overline{x_1}, \overline{x_2}\}\rangle \quad \text{and}$$
$$q_2 = \langle 5, \{y_2\}, \{\overline{x_3}, \overline{x_4}\}\rangle.$$

We can observe that this is indeed an $\mathcal{O}$-compatible set and that the first case of Theorem 5 is applicable. Hence, we will derive the clause $C_{\mathcal{Q}}$, which is $\overline{y_1} + \overline{y_2} \geq 1$.

[3] A more advanced example, as well as the proof for Theorem 5 can be found in the extended version of this paper (Vandesande, Coll, and Bogaerts 2025a).

Following Proposition 4, we can derive the clauses

$$C_{q_1} := \overline{y_1} + x_1 + x_2 \geq 1, \text{ and}$$
$$C_{q_2} := \overline{y_2} + x_3 + x_4 \geq 1$$

by Reverse Unit Propagation. Multiplying C_{q_1} and C_{q_2} by their respective weight and adding them up results in

$$3\overline{y_1} + 3x_1 + 3x_2 + 5\overline{y_2} + 5x_3 + 5x_4 \geq 8 \qquad (1)$$

Now, from the solution improving constraint, we can obtain

$$3x_1 + 3x_2 + 5x_3 + 5x_4 \leq 6 \qquad (2)$$

by the addition of literal axioms $x_4 \geq 0$ and $x_2 \geq 0$ multiplied by 2. Addition of (1) and (2), and simplification gives

$$3\overline{y_1} + 5\overline{y_2} \geq 2$$

Dividing this by 5 now yields $C_{\mathcal{Q}}$.

4 Certifying CNF Encodings based on (Multi-Valued) Decision Diagrams

When a solution is found (at line 11 of Algorithm 1), MAXCDCL decides heuristically whether or not to add a CNF-encoding of the solution-improving constraint

$$\mathcal{O} \leq v^* - 1.$$

For many CNF encodings of pseudo-Boolean constraints, it is well-known how to get certification (Vandesande 2023; Gocht et al. 2022; Berg et al. 2024a). One notable exception is the CNF encoding based on *binary decision diagrams* (BDD), which Bofill et al. (2020) recently generalized to so-called *multi-valued decision diagrams* (MDD) and is used by MAXCDCL. Most of the interesting questions from a proof logging perspective already show up for BDDs, so to keep the presentation more accessible, we focus on that case and afterwards briefly discuss how this generalizes to MDDs. Formal details and proofs are included in the supplementary material.

A *Binary Decision Diagram (BDD)* is a (node- and edge-)labeled graph with two leaves, labeled true ($\mathbf{t}$) and false ($\mathbf{f}$), respectively, where each internal node is labeled with a variable and has two outgoing edges, labeled true ($\mathbf{t}$) and false ($\mathbf{f}$), respectively. We write $\mathrm{child}(\eta, \mathbf{f})$ and $\mathrm{child}(\eta, \mathbf{t})$ for the children following the edge with labels $\mathbf{f}$ and $\mathbf{t}$, respectively. Each node η in a BDD represents a *Boolean function*: the true and false leaf nodes represent a tautology and contradiction respectively; if η is a node labeled x with true child $\eta_{\mathbf{t}}$ and false child $\eta_{\mathbf{f}}$, it maps any (total) assignment α to $\eta(\alpha)$, which is defined as $\eta_{\mathbf{t}}(\alpha)$ if $x \in \alpha$ and as $\eta_{\mathbf{f}}(\alpha)$ if $\overline{x} \in \alpha$.

A BDD is *ordered* if there is a total order of the variables such that each path through the BDD respects this order and it is *reduced* if two conditions hold: *(1)* no node has two identical children and *(2)* no two nodes have the same label, $\mathbf{t}$-child and $\mathbf{f}$-child. For a fixed variable ordering, each Boolean function has a unique ordered and reduced representation as a BDD, i.e., ordered and reduced BDDs form a canonical representation of Boolean functions.

When using BDDs to encode a solution-improving constraint $\mathcal{O} \leq v^* - 1$ as clauses, first a BDD representing this constraint is constructed, then a set of clauses is generated from this BDD.

Phase 1: Construction of a BDD The standard way to create a reduced and ordered BDD for

$$\sum_{i=1}^{n} v_i b_i \leq v^* - 1$$

is to first recursively descend and create BDDs $\eta_{\mathbf{t}}$ and $\eta_{\mathbf{f}}$ for

$$\sum_{i>1}^{n} v_i b_i \leq v^* - 1 - a_1 \quad \text{and}$$
$$\sum_{i>1}^{n} v_i b_i \leq v^* - 1$$

respectively, and then combine them into the node $\mathrm{bdd}(x_1, \eta_{\mathbf{t}}, \eta_{\mathbf{f}})$. Using memoization, the BDD is always kept reduced: if a node is being created with the same label, and the same two children as an existing node, the existing node is returned instead; if $\eta_{\mathbf{t}} = \eta_{\mathbf{f}}$, then $\eta_{\mathbf{t}}$ is returned.

This procedure, which first creates the two children, will however always take exponential time (in terms of the number of objective literals), while this can in many cases be avoided. Crucially, all PB constraints for which a BDD node are created are of the form $\sum_{i\geq k}^{n} v_i b_i \leq A$ and for a fixed k, the fact that we work with Boolean variables means that many different right-hand-sides will result in an equivalent constraint. Moreover, such values can be computed while constructing the BDD using a dynamic programming approach. In the following proposition, which formalizes the construction, we say that $[l, u]$ is a *degree interval* for $\sum_{i\geq k}^{n} v_i b_i$ when we mean that for all values A in the (possibly unbounded) interval $[l, u]$, $\sum_{i\geq k}^{n} v_i b_i \leq A$ is equivalent to $\sum_{i\geq k}^{n} v_i b_i \leq u$.

Proposition 7 ((Abío et al. 2012))**.** *If $[l_{\mathbf{t}}, u_{\mathbf{t}}]$ and $[l_{\mathbf{f}}, u_{\mathbf{f}}]$ are degree intervals for $\sum_{i=k+1}^{n} v_i b_i$, then*

$$[\max(l_{\mathbf{t}} + v_k, l_{\mathbf{f}}), \min(u_{\mathbf{t}} + v_k, u_{\mathbf{f}})]$$

is a degree interval for $\sum_{i=k}^{n} v_i b_i$.

The dynamic programming approach for creating ordered and reduced BDDs now consists in keeping track of this interval for each translated PB constraint and reusing already created BDDs whenever possible. From now on, we will identify a node in a BDD $\eta = \mathrm{bdd}(k, l, u)$ as the node that represents the boolean function $\sum_{i=k}^{n} v_i b_i \leq [l, u]$.

Phase 2: A CNF Encoding From the BDD Given a BDD that represents a PB constraint, constructed as discussed above, we can get a CNF encoding as follows.

- For each internal node η in the BDD, a new variable v_η is created; intuitively this variable is true only when the Boolean function represented by η is true. In practice for the two leaf nodes no variable is created but their truth value is filled in directly. However, in the proofs below we will, to avoid case splitting pretend that a variable exists for each node.
- For each internal node η that is labeled with literal b with children $\eta_{\mathbf{t}} = \mathrm{child}(\eta, \mathbf{t})$ and $\eta_{\mathbf{f}} = \mathrm{child}(\eta, \mathbf{f})$, the clauses

$$\overline{b} + v_{\eta_{\mathbf{t}}} + \overline{v}_\eta \geq 1, \quad \text{and} \tag{3}$$
$$v_{\eta_{\mathbf{f}}} + \overline{v}_\eta \geq 1 \tag{4}$$

  are added. The first clause expresses that if b is true and $\eta_{\mathbf{t}}$ is false, then so is η. The second clause expresses that whenever $\eta_{\mathbf{f}}$ is false, so is η (this does not hold for BDDs in general, but only for the specific ones generated here, where we know η to represent a constraint $\sum_{i\geq k}^{n} v_i b_i \leq A$ and $\eta_{\mathbf{f}}$ represents $\sum_{i\geq k+1}^{n} v_i b_i \leq A$, both with positive coefficients).
- Finally, for the top node $\eta_\top$ representing $\mathcal{O} \leq v^* - 1$, the unit clause $v_{\eta_\top}$ is added.

Certification Our strategy for proof logging for this encoding follows the general pattern described by Vandesande, De Wulf, and Bogaerts (2022), namely, we first introduce the fresh variables by explicitly stating what their meaning is in terms of pseudo-Boolean constraints and later we derive the clauses from those constraints. Due to the fact that BDDs are reduced, the first step becomes non-obvious. Indeed, the new variables represent multiple (equivalent) pseudo-Boolean constraints at once. To make this work in practice, for each node $\eta = \mathrm{bdd}(k, l, u)$, we will show we can derive the constraints that express

$$v_\eta \Rightarrow \sum_{i\geq k}^{n} v_i b_i \leq l \quad \text{and}$$
$$v_\eta \Leftarrow \sum_{i\geq k}^{n} v_i b_i \leq u.$$

We will refer to these two constraints as the *defining constraints* for node η. These constraints together precisely determine the meaning of v_η, but they also contain the information that the partial sum cannot take any values between $l+1$ and u. Hence, proving them is expected to require substantial effort. Once we have these constraints for each node, deriving the clauses becomes very easy.

Proposition 8. *For each internal node η, the clauses* (3) *and* (4) *can be derived by a cutting planes derivation consisting of one literal axiom, one multiplication, three additions and one division from the defining constraints for η,* $\mathrm{child}(\eta, \mathbf{t})$ *and* $\mathrm{child}(\eta, \mathbf{f})$.

Proposition 9. *Let $\eta_\top = \mathrm{bdd}(1, l, v^* - 1)$ for some $l \leq v^* - 1$ be the root node of the BDD. The unit clause $v_{\eta_\top} \geq 1$ can be derived by a cutting planes derivation of one addition and one deletion from the defining constraints for $\eta_\top$ and the solution-improving constraint.*

The harder part is to actually derive the defining constraints for the internal nodes. Also this can be done in the VERIPB proof system, as we show next.

Proposition 10. *Let $\eta = \mathrm{bdd}(k, l, u)$ be an internal node with label b_k and with children $\eta_{\mathbf{t}} = \mathrm{child}(\eta, \mathbf{t})$ and $\eta_{\mathbf{f}} = \mathrm{child}(\eta, \mathbf{f})$.*

If the defining constraints for $\eta_{\mathbf{t}}$ and $\eta_{\mathbf{f}}$ are given, then the defining constraints for η can be derived in the VeriPB proof system.

Proof Sketch. The first step is to introduce two reification variables v_η and v'_η as

$$v_\eta \Leftrightarrow \sum_{i \geq k}^{n} v_i b_i \leq l \quad \text{and}$$
$$v'_\eta \Leftrightarrow \sum_{i \geq k}^{n} v_i b_i \leq u.$$

The next step is showing that these two variables are actually the same; the difficult direction is showing that v'_η implies v_η. We do this by case splitting on b_k and deriving each case by contradiction using the defining constraints for $\eta_\mathbf{t}$ and $\eta_\mathbf{f}$. To derive $\overline{x} + \overline{v}'_\eta + v_\eta \geq 1$, assume that b_k and v'_η are true and that v_η is false. The assumption on b_k and v'_η together with the defining constraints for $\eta_\mathbf{t}$ makes that $v_{\eta_\mathbf{t}}$ is true; the assumption that b_k is true and v_η is false together with the defining constraints for $\eta_\mathbf{t}$ makes that $v_{\eta_\mathbf{t}}$ is false. This is clearly a contradiction. Using the defining constraints of $\eta_\mathbf{f}$, we can in a similar way derive $b_k + \overline{v}'_\eta + v_\eta \geq 1$. A short cutting planes derivation now allows us to derive that v'_η implies v_η and hence derive the required defining constraints. □

Finally, it can happen that due to reducedness of BDDs, a node actually represents multiple constraints. The following proposition shows that also this can be proven in VeriPB.

Proposition 11. *Consider a node $\eta = \mathrm{bdd}(k, l, u)$ and let k' and l' be numbers with $k' < k$ and $l' = l + \sum_{i=k'}^{k-1} v_i$ such that the Boolean functions $\sum_{i=k}^{n} v_i b_i \leq u$ and $\sum_{i=k'}^{n} v_i b_i \leq u$ are equivalent.*

From the defining constraints of $\mathrm{bdd}(k, l, u)$, there is a cutting planes derivation consisting of $6(k-k')+3$ steps that derives the defining constraints for $\mathrm{bdd}(k', l', u)$ for reification variable v_η.

Generalization to MDDs MaxCDCL not only makes use of BDDs for encoding the solution-improving constraint, but also of MDDs. The idea is as follows. In some cases, MaxCDCL can infer implicit at-most-one constraints. These are constraints of the form $\sum_{i \in I} b_i \leq 1$ where the b_i are literals in the objective $\mathcal{O}$. The detection of such constraints is common in MaxSAT solvers, and certification for it has been described by Berg et al. (2023), so we will not repeat this here. Now assume that a set of disjoint at-most-one constraints has been found. In an MDD, instead of branching on *single variable* in each node, we will branch on a set of variables for which an at-most-one constraint has previous been derived. This means a node does not have two, but $|I| + 1$ children: one for each variable in the set and one for the case where none of them is true. Otherwise, the construction and ideas remain the same. As far as *certification* is concerned: essentially all proofs continue to hold; the main difference is that the case splitting in the proof of Theorem 10 will now split into $|I| + 1$ cases instead of two and will then use the at most one constraint to derive that the conclusion must hold in exactly one of the cases.

5 Experiments

We implemented proof logging in the MaxCDCL solver as described by Coll et al. (2025b).[4] Each solver call was assigned a single core on a 2x32-core AMD EPYC 9384X. The time and memory limits are enforced by runlim.[5] As benchmarks, we used the instances of the MaxSAT Evaluation 2024 (Berg et al. 2024b). The implementation, together with the raw data from the experiments can be found on Zenodo (Vandesande, Coll, and Bogaerts 2025b).

Overhead in proof logging A scatter plot evaluating the overhead in the solving time due to proof logging is shown in Figure 1. The experiments ran with a time limit of 1 hour and a memory limit of 32GB. Out of the 701 instances that were solved without proof logging, six instances could not be solved with proof logging. Considering instances solved both with and without proof logging, the median overhead of proof logging is 19%, which shows that for most instances, proof logging overhead is manageable. However, we see that for 10% of the instances, MaxCDCL with proof logging takes more than 4.61 times the time to solve the instance using MaxCDCL without proof logging. This is in line with earlier work on certifying PB-to-CNF encodings using VeriPB, such as the work of Berg et al. (2024a). Preliminary tests indicate as possible explanation for this phenomenon that writing the defining constraints for a node in an MDD is linear in the number of objective literals. In practice, this leads to large overhead for instances with a large number of objective literals. Investigating possible ways to circumvent this is part of planned future work.

Overhead in proof checking While optimizing the checking time for the produced proofs was out of scope for this work, we also evaluated it by running the VeriPB proof checker (VeriPB) with a time limit of 10 hours and a memory limit of 64GB. Figure 2 shows a scatter plot comparing the solving time with proof logging with the proof checking time. Out of 695 produced proofs, 485 were checked successfully; for the other 210 instances, the proof checker ran out of time for 202 instances and out of memory for 8 instances. In the median proof verification took 42.94 times the solving time. We believe this checking overhead to be too high for making this usable in practice. However, there are several ways in which this can be improved. On the one hand, a new proof checker is currently under development for improved performance (PBOxide 2025). Simultaneously, there are ongoing efforts for building a proof *trimmer* for VeriPB proofs. The proofs generated by our tools can serve as benchmarks to further guide this ecosystem of VeriPB tools. On the other hand, once the proof checker is in a stable state, we can investigate which potential optimizations *to the generated proofs* benefit the proof checking time. One thing that comes to mind is using *RUP with hints*

[4] On top of the basic algorithm described above, the experiments include proof logging for a range of techniques included in MaxCDCL (Coll et al. 2025b), namely clause vivification, equivalent and failed literal detection, clause subsumption and simplification due to unit propagation.

[5] https://github.com/arminbiere/runlim

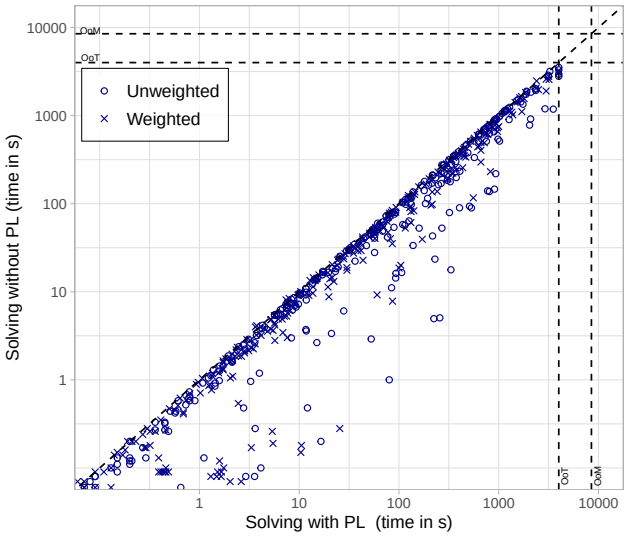

Figure 1: Comparison of solving time with and without proof logging.

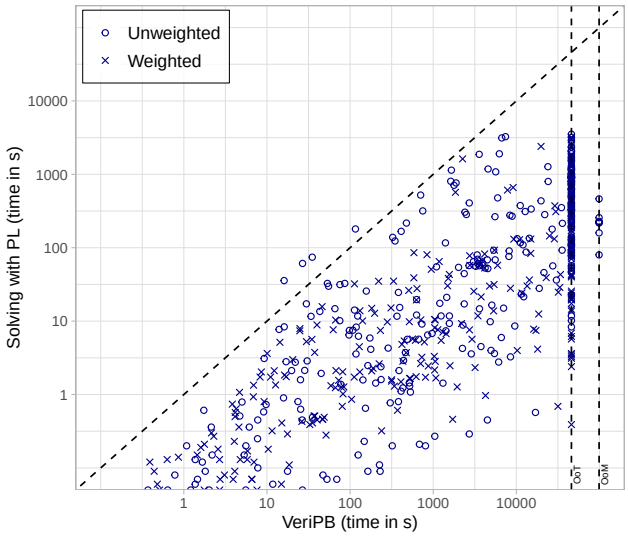

Figure 2: Comparison of solving time (with proof logging enabled) with proof checking time.

instead of plain RUP, so that the checker does not need to perform unit propagation.[6]

6 Conclusion

In this paper, we demonstrate how to add proof logging to the branch-and-bound MaxSAT solving paradigm using the VeriPB proof system. To do so, we formalize the paradigm with proof logging in mind, offering a new perspective on how these solvers operate. To put our formalization to the test, we implemented proof logging in the state-of-the-art branch-and-bound solver MaxCDCL, including the MDD encoding of the solution-improving search constraint, which is created by MaxCDCL to improve its performance. The main challenge in certifying the MDD encoding turns out to be proving that a node in an MDD represents multiple equivalent Boolean functions. Experiments show that proof logging overhead is generally manageable, while proof checking performance still leaves room for improvement.

As future work, we plan to add proof logging to literal unlocking, a recent technique (Li et al. 2025) that improves lower bounds during look-ahead.

Acknowledgments

This work is partially funded by the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

In addition, this work is partially funded by the Fonds Wetenschappelijk Onderzoek – Vlaanderen (projects G064925N and G070521N & fellowship 11A5J25N), and by grants PID2021-122274OB-I00 and PID2024-157625OB-I00 funded by MICIU/AEI/10.13039/501100011033/FEDER, UE.

[6]For more information on these rules, see the VeriPB repository (VeriPB).

References

Abío, I.; Nieuwenhuis, R.; Oliveras, A.; Rodríguez-Carbonell, E.; and Mayer-Eichberger, V. 2012. A New Look at BDDs for Pseudo-Boolean Constraints. *J. Artif. Intell. Res.*, 45: 443–480.

Abramé, A.; and Habet, D. 2014. Ahmaxsat: Description and Evaluation of a Branch and Bound Max-SAT Solver. *J. Satisf. Boolean Model. Comput.*, 9(1): 89–128.

Berg, J.; Bogaerts, B.; Nordström, J.; Oertel, A.; Paxian, T.; and Vandesande, D. 2024a. Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability. In *CP*, volume 307 of *LIPIcs*, 4:1–4:28. Schloss Dagstuhl.

Berg, J.; Bogaerts, B.; Nordström, J.; Oertel, A.; and Vandesande, D. 2023. Certified Core-Guided MaxSAT Solving. In *CADE*, volume 14132 of *LNCS*, 1–22. Springer.

Berg, J.; Järvisalo, M.; Martins, R.; Niskanen, A.; and Paxian, T. 2024b. MaxSAT Evaluation 2024. https://maxsat-evaluations.github.io/2024/.

Bertot, Y.; and Castéran, P. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Biere, A. 2006. TraceCheck. http://fmv.jku.at/tracecheck/.

Bleukx, I.; Flippo, M.; Demirović, E.; Bogaerts, B.; and Guns, T. 2026. Using Certifying Constraint Solvers for Generating Step-wise Explanations. In *Proceedings of The 40th Annual AAAI Conference on Artificial Intelligence*. Accepted for publication.

Bofill, M.; Coll, J.; Suy, J.; and Villaret, M. 2020. An MDD-based SAT encoding for pseudo-Boolean constraints with at-most-one relations. *Artif. Intell. Rev.*, 53(7): 5157–5188.

Bogaerts, B.; Gocht, S.; McCreesh, C.; and Nordström, J. 2023. Certified Dominance and Symmetry Breaking for Combinatorial Optimisation. *J. Artif. Intell. Res.*, 77: 1539–1589.

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

Brummayer, R.; and Biere, A. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the SMT'09*, 1–5. ISBN 9781605584843.

Brummayer, R.; Lonsing, F.; and Biere, A. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *SAT*, volume 6175 of *LNCS*, 44–57. Springer.

Coll, J.; Li, C. M.; Li, S.; Habet, D.; and Manyà, F. 2025a. Solving weighted Maximum Satisfiability with Branch and Bound and clause learning. *Comput. Oper. Res.*, 183: 107195.

Coll, J.; Li, C.-M.; Li, S.; Habet, D.; and Manyà, F. 2025b. Solving weighted maximum satisfiability with branch and bound and clause learning. *Computers & Operations Research*, 107195.

Cook, W. J.; Coullard, C. R.; and Turán, G. 1987. On the complexity of cutting-plane proofs. *Discret. Appl. Math.*, 18(1): 25–38.

Cook, W. J.; Koch, T.; Steffy, D. E.; and Wolter, K. 2013. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Math. Program. Comput.*, 5(3): 305–344.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt, W. A., Jr.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient Certified RAT Verification. In *CADE*, volume 10395 of *LNCS*, 220–236. Springer.

de Moura, L.; and Ullrich, S. 2021. The Lean 4 Theorem Prover and Programming Language. In *CADE*, volume 12699 of *LNCS*, 625–635. Springer.

Ferreira, N. G.; and Silva, P. S. M. 2004. Automatic Verification of Safety Rules for a Subway Control Software. In *SBMF*, volume 130 of *Electronic Notes in Theoretical Computer Science*, 323–343. Elsevier.

Fleury, M. 2020. *Formalization of logical calculi in Isabelle/HOL*. Ph.D. thesis, Saarland University, Saarbrücken, Germany.

Gange, G.; Stuckey, P. J.; and Szymanek, R. 2011. MDD propagators with explanation. *Constraints An Int. J.*, 16(4): 407–429.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. SolverCheck: Declarative Testing of Constraints. In *CP*, volume 11802 of *LNCS*, 565–582. Springer.

Gocht, S.; Martins, R.; Nordström, J.; and Oertel, A. 2022. Certified CNF Translations for Pseudo-Boolean Solving. In *SAT*, volume 236 of *LIPIcs*, 16:1–16:25. Schloss Dagstuhl.

Gocht, S.; and Nordström, J. 2021. Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs. In *AAAI*, 3768–3777. AAAI Press.

Goldberg, E. I.; and Novikov, Y. 2003. Verification of Proofs of Unsatisfiability for CNF Formulas. In *DATE*, 10886–10891. IEEE Computer Society.

Heras, F.; Larrosa, J.; and Oliveras, A. 2008. MiniMaxSAT: An Efficient Weighted Max-SAT solver. *J. Artif. Intell. Res.*, 31: 1–32.

Heule, M.; Hunt, W. A., Jr.; and Wetzler, N. 2013. Trimming while checking clausal proofs. In *FMCAD*, 181–188. IEEE.

Heule, M.; Järvisalo, M.; Suda, M.; Iser, M.; and Balyo, T. 2022. The 2022 International SAT Competition. https://satcompetition.github.io/2022/.

Ihalainen, H.; Vandesande, D.; Schidler, A.; Berg, J.; Bogaerts, B.; and Järvisalo, M. 2026. Efficient and Reliable Hitting-Set Computations for the Implicit Hitting Set Approach. In *Proceedings of The 40th Annual AAAI Conference on Artificial Intelligence*. Accepted for publication.

Jabs, C.; Berg, J.; Bogaerts, B.; and Järvisalo, M. 2025. Certifying Pareto Optimality in Multi-Objective Maximum Satisfiability. In *TACAS (2)*, volume 15697 of *LNCS*, 108–129. Springer.

Järvisalo, M.; Berg, J.; Martins, R.; and Niskanen, A. 2023. MaxSAT Evaluation 2023. https://maxsat-evaluations.github.io/2023/.

Järvisalo, M.; Heule, M.; and Biere, A. 2012. Inprocessing Rules. In *IJCAR*, volume 7364 of *LNCS*, 355–370. Springer.

Kügel, A. 2010. Improved Exact Solver for the Weighted MAX-SAT Problem. In *POS@SAT*, volume 8 of *EPiC Series in Computing*, 15–27. EasyChair.

Larrosa, J.; Nieuwenhuis, R.; Oliveras, A.; and Rodríguez-Carbonell, E. 2011. A Framework for Certified Boolean Branch-and-Bound Optimization. *J. Autom. Reason.*, 46(1): 81–102.

Leivo, M.; Berg, J.; and Järvisalo, M. 2020. Preprocessing in Incomplete MaxSAT Solving. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, 347–354. IOS Press.

Li, C.; Xu, Z.; Coll, J.; Manyà, F.; Habet, D.; and He, K. 2021. Combining Clause Learning and Branch and Bound for MaxSAT. In *CP*, volume 210 of *LIPIcs*, 38:1–38:18. Schloss Dagstuhl.

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

Li, C. M.; and Manyà, F. 2021. MaxSAT, Hard and Soft Constraints. In *Handbook of Satisfiability*, volume 336 of *FAIA*, 903–927. IOS Press.

Li, C. M.; Manyà, F.; and Planes, J. 2007. New Inference Rules for Max-SAT. *J. Artif. Intell. Res.*, 30: 321–359.

Li, S.; Li, C.; Coll, J.; Habet, D.; and Manyà, F. 2025. Improving the Lower Bound in Branch-and-Bound Algorithms for MaxSAT. In *AAAI*, 11272–11281. AAAI Press.

Manlove, D. F.; and O'Malley, G. 2014. Paired and Altruistic Kidney Donation in the UK: Algorithms and Experimentation. *ACM J. Exp. Algorithmics*, 19(1).

McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P. 2011. Certifying algorithms. *Comput. Sci. Rev.*, 5(2): 119–161.

Morgado, A.; and Marques-Silva, J. 2011. On Validating Boolean Optimizers. In *ICTAI*, 924–926. IEEE Computer Society.

Nipkow, T.; Paulson, L. C.; and Wenzel, M. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

Nogueira, M. L.; Balduccini, M.; Gelfond, M.; Watson, R.; and Barry, M. 2001. An A-Prolog Decision Support System for the Space Shuttle. In *PADL*, volume 1990 of *LNCS*, 169–183. Springer.

PBOxide 2025. 2025. PBOxide: Verifier for pseudo-Boolean proofs rewritten in Rust. https://gitlab.com/MIAOresearch/software/pboxide.

Py, M.; Cherif, M. S.; and Habet, D. 2020. Towards Bridging the Gap Between SAT and Max-SAT Refutations. In *ICTAI*, 137–144. IEEE.

Py, M.; Cherif, M. S.; and Habet, D. 2022. Proofs and Certificates for Max-SAT. *J. Artif. Intell. Res.*, 75: 1373–1400.

Silva, J. P. M.; and Sakallah, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5): 506–521.

Slind, K.; and Norrish, M. 2008. A Brief Overview of HOL4. In *TPHOLs*, volume 5170 of *LNCS*, 28–32. Springer.

Vandesande, D. 2023. *Towards Certified MaxSAT Solving: Certified MaxSAT solving with SAT oracles and encodings of pseudo-Boolean constraints*. Master's thesis, Vrije Universiteit Brussel (VUB).

Vandesande, D.; Coll, J.; and Bogaerts, B. 2025a. Certified Branch-and-Bound MaxSAT Solving (Extended Version). arXiv:2511.10273.

Vandesande, D.; Coll, J.; and Bogaerts, B. 2025b. Experimental Repository for "Certified Branch-and-Bound MaxSAT Solving" (AAAI26). https://doi.org/10.5281/zenodo.7709687.

Vandesande, D.; De Wulf, W.; and Bogaerts, B. 2022. QMaxSATpb: A Certified MaxSAT Solver. In *LPNMR*, volume 13416 of *LNCS*, 429–442. Springer.

VeriPB. 2024. VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB.

Wetzler, N.; Heule, M.; and Hunt, W. A., Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *SAT*, volume 8561 of *LNCS*, 422–429. Springer.