Certified Core-Guided MaxSAT Solving

Jeremias Berg[1] $^{[0000-0001-7660-8061]}$, Bart Bogaerts[2] $^{[0000-0003-3460-4251]}$, Jakob Nordström[3,4] $^{[0000-0002-2700-4285]}$, Andy Oertel[4,3] $^{[0000-0001-9783-6768]}$ ✉, and Dieter Vandesande[2] $^{[0000-0002-8150-3202]}$

[1] HIIT, Department of Computer Science, University of Helsinki, Helsinki, Finland
[2] Vrije Universiteit Brussel, Artificial Intelligence Laboratory, Brussels, Belgium
[3] University of Copenhagen, Copenhagen, Denmark
[4] Lund University, Lund, Sweden
andy.oertel@cs.lth.se

Abstract. In the last couple of decades, developments in SAT-based optimization have led to highly efficient maximum satisfiability (MaxSAT) solvers, but in contrast to the SAT solvers on which MaxSAT solving rests, there has been little parallel development of techniques to prove the correctness of MaxSAT results. We show how pseudo-Boolean proof logging can be used to certify state-of-the-art core-guided MaxSAT solving, including advanced techniques like structure sharing, weight-aware core extraction and hardening. Our experimental evaluation demonstrates that this approach is viable in practice. We are hopeful that this is the first step towards general proof logging techniques for MaxSAT solvers.

Keywords: MaxSAT · core-guided search · proof logging · certifying algorithms

1 Introduction

Combinatorial optimization is one of the most impressive, and most intriguing, success stories in computer science. This area deals with computationally very challenging problems, which are widely believed to require exponential time in the worst case [21, 49]. In spite of this, during the last couple of decades astonishing progress has been made on so-called combinatorial solvers for a number of different algorithmic paradigms such as Boolean satisfiability (SAT) solving and optimization [15], constraint programming (CP) [72], and mixed integer programming (MIP) [1, 16]. Today, such solvers are routinely used to solve real-world problems with hundreds of thousands or even millions of variables.

While the performance of modern combinatorial solvers is truly impressive, one negative aspect is that they are highly complex pieces of software, and it is well documented that even mature state-of-the-art solvers sometimes give wrong results [2, 18, 25, 37]. This can be fatal for applications where correctness is a non-negotiable demand. Perhaps the most successful approach for addressing this problem so far is the requirement in the SAT solving community that solvers should be *certifying* [3, 62], meaning that when given a formula a solver should output not only a verdict whether the formula is satisfiable or unsatisfiable, but also an efficiently machine-verifiable *proof log* establishing that this verdict is guaranteed to be correct. One can then feed the input formula, the verdict, and the proof log to a special, dedicated *proof checker*, and accept the result if the proof checker agrees that the proof log shows that the solver computation is valid. Over the years, different proof formats such as *RUP* [43], *TraceCheck* [14], *DRAT* [44,45], *GRIT* [27], and *LRAT* [26] have been developed, and for almost a decade *DRAT* proof logging has been compulsory in the (main track of the) SAT competition. However, there has been very limited progress in designing analogous proof logging techniques for more powerful algorithmic paradigms.

Our focus in this work is on the optimization paradigm that is arguably closest to SAT solving, namely *maximum satisfiability* or *MaxSAT* solving [8,56], and the challenge of developing proof logging techniques for MaxSAT solvers.

1.1 Previous Work

Since essentially all modern MaxSAT solvers are based on repeated invocations of SAT solvers, a first question is why SAT proof logging techniques are not sufficient. While *DRAT* is a very powerful proof system, it seems that the overhead of generating proofs of correctness for the rewriting steps in between SAT solver calls in MaxSAT solvers is too large to be tolerable for practical purposes. Another, related, problem is that for optimization problems one needs to reason about the objective function, which *DRAT* struggles to do since its language is limited to disjunctive clauses. But perhaps the biggest challenge is that while modern SAT solving is completely dominated by the *conflict-driven clause learning (CDCL)* method [11,59,66], for MaxSAT there is a rich variety of approaches including *linear SAT-UNSAT* (or *model-improving search*) [31,54,68], *core-guided search* [4,7,35,67], *implicit hitting set (IHS)* search [28,29], and some recent work on branch-and-bound methods [57] (where we stress that the lists of references are far from exhaustive).

One tempting solution to circumvent this heterogeneity of solving approaches is to treat the MaxSAT solver as a black box and use a single call to a certifying SAT solver to prove optimality of the final solution found. However, there are several problems with this proposal. Firstly, we would still need proof logging to ensure that the input to the SAT solver is a correct encoding of a claim of optimality for the correct problem instance. Secondly, such a SAT call could be extremely expensive, running counter to the goal of proof logging with low (and predictable) overhead. Finally, even if the SAT-call approach could be made to work efficiently, this would just certify the final result, and would not help validate the correctness of the reasoning of the solver. For these reasons, our goal is to provide proof logging for the actual computations of the MaxSAT algorithm.

While some proof systems and tools have been developed specifically for MaxSAT [19,34,48,53,64,65,69,70,71], none of them comes close to providing general-purpose proof logging, because they apply only for very specific algorithm implementations and/or fail to capture the full range of reasoning used in an algorithmic approach. A recent work [75] by two co-authors on the current paper instead leverages the pseudo-Boolean proof logging system VeriPB [76] to certify correctness of the unweighted linear SAT-UNSAT solver QMaxSAT. VeriPB is similar in spirit to *DRAT*, but operates with more general 0–1 linear inequalities rather than just clauses. This simplifies reasoning about optimization problems, and also makes it possible to capture the powerful MaxSAT solver inferences in a more concise way. VeriPB has previously been used for proof logging of enhanced SAT solving techniques [17, 42] and pseudo-Boolean solving [38], as well as for providing proof-of-concept tools for a nontrivial range of techniques in constraint programming [33, 41] and subgraph solving [39, 40].

1.2 Our Contributions

In this work, we use VeriPB to provide, to the best of our knowledge for the first time, efficient proof logging for the full range of techniques in a cutting-edge MaxSAT solver. We consider the state-of-the-art core-guided solver CGSS [47], based on RC2 [46], and show how to enhance CGSS to output proofs of correctness of its reasoning, including sophisticated techniques such as stratification [6, 58], intrinsic-at-most-one constraints [46], hardening [6], weight-aware core-extraction [13], and structure sharing [47]. We find that the overhead for such proof logging is perfectly manageable, and although there is certainly room to improve the proof verification time, our experiments demonstrate that already a first proof-of-concept implementation of this approach is practically feasible.

It has been shown previously [32, 39, 52] that proof logging can also serve as a powerful debugging tool. This is because faulty reasoning is likely to lead to unsound proofs, which can be detected even if the solver produces correct output for all test cases. We exhibit yet another example of this—some proofs for which we struggled to make the verification work turned out to reveal two well-hidden bugs in RC2 and CGSS that earlier extensive testing had failed to uncover.

Although it still remains to provide proof logging for other MaxSAT approaches such as (general, weighted) linear SAT-UNSAT and implicit hitting set (IHS) search, we are optimistic that our work could serve as an important step towards general adoption of proof logging techniques for MaxSAT solvers.

1.3 Outline of This Paper

After reviewing preliminaries for pseudo-Boolean reasoning and core-guided MaxSAT solving in Sections 2 and 3, we explain how core-guided MaxSAT solvers can be equipped with proof logging methods in Section 4. In Section 5 we present our experimental evaluation, after which some concluding remarks and directions for future research are given in Section 6.

2 Preliminaries

We start by a review of some standard material which can be found, e.g., in [20, 38, 42]. A *literal* ℓ over a Boolean variable x (taking values in $\{0, 1\}$, which we identify with false and true, respectively) is x itself or its negation $\overline{x}$, where $\overline{x} = 1 - x$. A *pseudo-Boolean (PB)* constraint is a 0-1 integer linear inequality $C \doteq \sum_i a_i \ell_i \geq A$ (where $\doteq$ denotes syntactic equality). When convenient, we can assume without loss of generality that PB constraints are in *normalized form* [10]; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative integers. The set of literals in C is denoted *lits*(C). The *negation* of C is $\neg C \doteq \sum_i a_i \ell_i \leq A - 1$ (rewritten in normalized form when needed). A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints. Note that a disjunctive clause can be viewed as a PB constraint with all coefficients and the degree equal to 1, and so formulas in conjunctive normal form (CNF) are special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{0, 1\}$, which we extend to literals by respecting the meaning of negation. Applying ρ to a constraint C yields $C{\restriction_\rho}$ by substituting the variables assigned in ρ by their values, and for a formula $F \doteq \bigwedge_j C_j$ we define $F{\restriction_\rho} \doteq \bigwedge_j C_j{\restriction_\rho}$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$, and ρ satisfies F if it satisfies all $C \in F$, in which case F is *satisfiable*. A formula lacking satisfying assignments is *unsatisfiable*. We say that F *implies* C, denoted $F \models C$, if any assignment satisfying F also satisfies C.

An *objective* $O \doteq \sum_i w_i \ell_i + M$ is an affine function over literals ℓ_i to be minimized by (total) assignments α satisfying F. The *value* (or *cost*) of an objective O under such an α, which we refer to as a *solution*, is $O(\alpha) = \sum_{\alpha(\ell_i)=1} w_i + M$. We write *coeff*$(O, \ell_i)$ to denote the coefficient w_i of a literal $\ell_i \in$ *lits*(O).

The foundation of the pseudo-Boolean proof logging in this paper is the *cutting planes* proof system [24], which is a method to iteratively derive new constraints implied by a pseudo-Boolean formula F. If C and D have been derived before or are *axiom constraints* in F, then any positive *linear combination* of these constraints can be derived. *Literal axioms* $\ell \geq 0$ can also be added to any previously derived constraints. For a constraint $\sum_i a_i \ell_i \geq A$ in normalized form, *division* by a positive integer d derives $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, and we also add a *saturation* rule that derives $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ (where the soundness of these rules crucially depends on the normalized form). It is well known that any PB constraint implied by F can be derived using these rules.

A constraint C is said to *unit propagate* the literal ℓ to true under an assignment ρ if $C{\restriction_\rho}$ cannot be satisfied unless ℓ is true. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating or some constraint C wants to propagate a literal that has already been assigned to the opposite value. The latter case is called a *conflict*, since C is *violated* by ρ'. We say that F implies C by *reverse unit propagation (RUP)*, and that C is a *RUP constraint* with respect to F, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $F \models C$ holds if C is a RUP constraint, and as a convenient shorthand we will add a RUP rule for deriving new constraints.

In addition to deriving constraints that are implied by a formula F, we also allow deriving so-called *redundant* constraints C that are *not* implied by F as long as some optimal solution is guaranteed to be preserved. This is done by extending the proof system with a *redundance-based strengthening* rule [17,42]. We will only need the special case of this rule saying that for a fresh variable z and for any constraint $D \doteq \sum_i a_i \ell_i \geq A$ we can introduce the *reified constraints*

$$C^{\Rightarrow}_{\text{reif}}(z, D) \doteq A\overline{z} + \textstyle\sum_i a_i \ell_i \geq A \tag{1a}$$

$$C^{\Leftarrow}_{\text{reif}}(z, D) \doteq \left(\textstyle\sum_i a_i - A + 1\right) z + \textstyle\sum_i a_i \overline{\ell}_i \geq \textstyle\sum_i a_i - A + 1 \tag{1b}$$

encoding the implications $z \Rightarrow D$ and $z \Leftarrow D$, respectively. We refer to z as the *reification variable*, and when D is clear from context, we will sometimes write just $C^{\Rightarrow}_{\text{reif}}(z)$ for (1a) and $C^{\Leftarrow}_{\text{reif}}(z)$ for (1b).

The *maximum satisfiability (MaxSAT) problem* can be described conveniently as a special case of pseudo-Boolean optimization. A discussion on the equivalence of the following and the—more classical—clause-centric definition can be found in, for instance, [8,55]. An instance (F, O) of the (weighted partial) MaxSAT problem consists of a CNF formula F and an objective function O written as a non-negative affine combination of literals. The goal is to find a solution α that satisfies F and minimizes $O(\alpha)$. We say that such a solution α is *optimal* for the instance and that the optimal cost of the instance (F, O) is $O(\alpha)$.

3 The OLL Algorithm for Core-Guided MaxSAT Solving

We now proceed to discuss the core-guided MaxSAT solving in CGSS, which is based on the OLL algorithm [5,63], and describe the main heuristics used in efficient implementations of this algorithm. Given a MaxSAT instance (F_{orig}, O_{orig}), OLL takes an optimistic view and attempts to find an assignment satisfying F_{orig} in which O_{orig} equals its constant term (i.e., all literals in $\mathit{lits}(O_{orig})$ are false). If such a solution exists, it is clearly optimal. Otherwise, the solver will extract a *core* K, which is a clause such that (i) K only contains objective literals, i.e., $\mathit{lits}(K) \subseteq \mathit{lits}(O_{orig})$, and (ii) F_{orig} implies K, which means that any solution to F_{orig} has to set at least one literal in $\mathit{lits}(K)$ to true. The *cost* $w(K, O) = \min\{\mathit{coeff}(O, \ell) : \ell \in \mathit{lits}(K)\}$ of a core K is the smallest coefficient in the objective O of any literal in K. The core K is used to (conceptually) reformulate the instance into (F_{ref}, O_{ref}) which has the same minimal-cost solutions. The constant term LB in O_{ref} is a lower bound on the optimal cost of the instance, and the reformulation is done in such a way that the lower bound increases (exactly) with the cost of the core K as defined above.

In more detail, the algorithm maintains a reformulated objective O_{ref} (initialized to O_{orig}) such that the (non-normalized) pseudo-Boolean constraint

$$O_{orig} \geq O_{ref} \doteq \sum_{b \in \mathit{lits}(O_{orig})} \mathit{coeff}(O_{orig}, b) \cdot b \geq \sum_{b' \in \mathit{lits}(O_{ref})} \mathit{coeff}(O_{ref}, b') \cdot b' + LB \tag{2}$$

is satisfied by all solutions of F_{ref}. Note that the constraint (2), which we refer to as an *objective reformulation constraint*, implies that the constant term LB is a lower bound on the optimal cost.

In each iteration, a SAT solver is queried for a solution α to F_{ref} with $O_{ref}(\alpha) = LB$. If such an α exists, the constraint (2) yields that $O_{orig}(\alpha) = LB$, and so α is a minimal-cost solution to (F_{orig}, O_{orig}). Otherwise, the solver returns a new core K that requires at least one literal in $lits(O_{ref})$ to be set to 1. This implies that the optimal cost is strictly larger than LB, and the core K is used for a new reformulation step.

The objective reformulation step adds new clauses to F_{ref} encoding the constraints $y_{K,k} \Leftarrow \sum_{b \in Lit(K)} b \geq k$ for $k = 2, \ldots, |K|$. The new variables $y_{K,k}$ are added to O_{ref} with coefficient $w(K, O_{ref})$ equalling the cost of K, and the coefficient in O_{ref} of each literal in K is decreased by the same amount. Finally, the lower bound LB—the constant term of O_{ref}—is also increased by $w(K, O_{ref})$. Since $y_{K,k}$ encodes that at least k literals in K are true, we have the equality $\sum_{b \in lits(K)} b = 1 + \sum_{k=2}^{|K|} y_{K,k}$, where the additive 1 comes from the fact that at least one literal in K has to be true, and the reformulation step is just applying this equality multiplied by $w(K, O_{ref})$ to O_{ref}. Notice that the variables added during objective reformulation can later be discovered in other cores. In practice, all implementations of OLL we are aware of encode the semantics of counting variables incrementally [60]. This means that initially only the variable $y_{K,2}$ is defined, and the variable $y_{K,i+1}$ is introduced only after $y_{K,i}$ is found in a core.

Implementations of OLL for MaxSAT—including the CGSS solver that we enhance with proof logging in this work—extend the algorithm with a number of heuristics such as stratification [6, 58], hardening [6], the intrinsic-at-most-ones technique [46], weight-aware core extraction [13], and structure sharing [47].

Stratification extracts cores not over all literals in O_{ref} but only over those whose coefficient is above some bound w_{strat}. This steers search toward cores containing literals with high coefficients, resulting in larger increases of LB. Once no more cores over such variables can be found, the algorithm lowers w_{strat}, terminating only after no more cores can be found with $w_{strat} = 1$. The fact that no more cores containing only variables with coefficients above w_{strat} exist is detected by the SAT solver returning a (possibly non-optimal) solution α. The minimal cost $O_{orig}(\alpha)$ of all such solutions gives an upper bound UB on the optimal cost of the instance, allowing OLL to terminate as soon as $LB = UB$.

Hardening fixes literals in O_{ref} to 0 based on information provided by the current upper and lower bounds UB and LB. If for any $b \in lits(O_{ref})$ it holds that $coeff(O_{ref}, b) + LB > UB$, then any solution α with $b = 1$ would have higher cost than the current best solution known, and would thus not be optimal.

The *intrinsic-at-most-one* technique identifies subsets $\mathcal{S} \subseteq lits(O_{ref})$ of objective literals such that $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ is implied, i.e., any solution can assign at most one literal in $\mathcal{S}$ to 0. This is used both to increase the lower bound and to reformulate the objective. If we let $w_{min} = \min\{coeff(O_{ref}, b) : b \in \mathcal{S}\}$, then $\mathcal{S}$ implies a lower bound increase of $LB_{\mathcal{S}} = (|\mathcal{S}| - 1) \cdot w_{min}$. Additionally, we define a new variable $\ell_{\mathcal{S}}$ by the clause $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ to indicate if in fact all literals in $\mathcal{S}$ are true, and introduce it in the reformulated objective with coefficient w_{min}. This means that we remove the already known lower bound $LB_{\mathcal{S}}$ from O_{ref} and transfer the possible additional cost w_{min} from $\mathcal{S}$ to the variable $\ell_{\mathcal{S}}$.

Weight-aware core extraction (WCE) delays objective reformulation, and the accompanying increase in new variables and clauses, for as long as possible. When a new core K is extracted by a solver that uses WCE, initially only the coefficient of each $b \in \mathit{lits}(K)$ is lowered and the lower bound LB is increased by $w(K, O_{\mathit{ref}})$. Then the SAT solver is invoked again with the literals, that still have coefficients above w_{strat} in O_{ref}, set to 0. When the SAT solver finds a satisfying assignment extending the assumptions, all objective reformulations steps are then performed at once. This is correct since the final effect is the same as if the core would have been discovered one by one and immediately followed by objective reformulation. Notice that this core extraction loop is guaranteed to terminate since the coefficient of at least one variable is decreased to 0 for each new core. *Structure sharing* is a recent extension to weight-aware core extraction that makes use of the potential overlap in cores detected in order to achieve more compact encodings of counting variable semantics.

4 Proof Logging for the OLL Algorithm for MaxSAT

We have now reached a point where we can describe the contribution of this work, namely how to add proof logging to an OLL-based core-guided MaxSAT solver, including all the state-of-the-art techniques described in Section 3.

In our proof logging routines we maintain the invariants described next. The reformulated objective O_{ref} is already implicitly tracked by the solver and at all times it is possible to derive that $O_{\mathit{orig}} \geq O_{\mathit{ref}}$ as in (2). We also keep track of the current upper bound UB on O_{orig} and best solution α_{best} found so far. All cores that have been found and processed are in the set $\mathcal{K}$.

SAT Solver Calls. The CDCL SAT solvers used in core-guided MaxSAT algorithms can support *DRAT* proof logging, and since the proof format used by VeriPB is a strict extension of *DRAT* (modulo small and purely syntactical modifications) it is straightforward to provide proof logging for the part of the reasoning done in SAT solver calls, and to add all learned clauses to the proof checker database.

Each invocation of the SAT solver returns either a new solution α or a new core K. When a solution α with $O_{\mathit{orig}}(\alpha) < \mathit{UB}$ is obtained, it is logged in the proof, which adds the *objective-improving constraint*

$$O_{\mathit{orig}} \leq \mathit{UB} - 1 \tag{3a}$$

(which is

$$\sum_{b \in \mathit{lits}(O_{\mathit{orig}})} \mathit{coeff}(O_{\mathit{orig}}, b) \cdot \bar{b} \;\geq\; 1 + \sum_{b \in \mathit{lits}(O_{\mathit{orig}})} \mathit{coeff}(O_{\mathit{orig}}, b) - \mathit{UB} \tag{3b}$$

in normalized form). A technical side remark is that later solutions with cost greater than UB cannot successfully be logged, since they violate the constraint (3a) added to the proof checker database, and so the proof logging routines make sure to only log solutions that improve the current upper bound.

If the SAT solver instead returns a new core K, this clause is guaranteed to be a reverse unit propagation (RUP) clause with respect to the set of clauses currently in the solver database, and so we can use the RUP rule to add K to the proof checker database (which contains a superset of the clauses known by the solver). For our book-keeping, we also add K to the set $\mathcal{K}$. A special case is that K could be the contradictory empty clause, corresponding to the pseudo-Boolean constraint $0 \geq 1$. This means that there are no solutions to the formula.

To optimize the efficiency of proof verification, constraints should be deleted from the proof when they are no longer needed. Since SAT solver proofs are only used to prove *unsatisfiability* this does not cause any issues, but when certifying *optimality* we have to be careful in order not to create better-than-optimal solutions (which could happen if, e.g., constraints in the input formula are removed). The *checked deletion* rule [17] ensuring this in VeriPB does not have any analogue in *DRAT*, so some care is needed here when translating SAT solver proofs into the VeriPB format.

Incremental Totalizer with Structure Sharing. Different implementations of OLL for MaxSAT differ in which encoding is used for the counting variables introduced during objective reformulation [9, 50, 51]. The two solvers we consider use totalizers [9], so we start by explaining this encoding and then show how to provide proof logging for the clauses added to the proof checker database.

The totalizer encoding for a set $I = \{\ell_1, \ldots, \ell_n\}$ of literals is a CNF formula $\mathcal{T}$ that defines *counting variables* $y_{I,j}$ for $j = 1, \ldots, n$ such that for any assignment that satisfies $\mathcal{T}$ the variable $y_{I,j}$ is true if and only if $\sum_{i=1}^{n} \ell_i \geq j$. The structure of $\mathcal{T}$ can be viewed as a binary tree, with literals in I at the leaves and with each internal node η associated with variables counting the true leaf literals in the subtree rooted at η. The variables $y_{I,j}$ are associated with the root of the tree.

More formally, given a set of literals I, we construct a binary tree with leaves labelled by the literals in I. For every node η of $\mathcal{T}$, let $\mathit{lits}(\eta)$ denote the leaves in the subtree rooted at η; where it is convenient, we will overload I to also refer to the root note. For each internal node η, the totalizer encoding introduces the counting variables $S_\eta = \{y_{\eta,1}, \ldots, y_{\eta,|\mathit{lits}(\eta)|}\}$, the meaning of which can be encoded recursively in terms of the variables S_{η_1} and S_{η_2} for the children η_1 and η_2 of η by the (pseudo-Boolean form of the) clauses

$$C_\eta^{\Leftarrow}(\alpha, \beta, \sigma) \doteq y_{\eta,\sigma} + \overline{y}_{\eta_1,\alpha} + \overline{y}_{\eta_2,\beta} \geq 1 \tag{4a}$$

$$C_\eta^{\Rightarrow}(\alpha, \beta, \sigma) \doteq \overline{y}_{\eta,\sigma+1} + y_{\eta_1,\alpha+1} + y_{\eta_2,\beta+1} \geq 1 \tag{4b}$$

for all integers α, β, σ such that $\alpha + \beta = \sigma$ and $0 \leq \alpha \leq |\mathit{lits}(\eta_1)|$, $0 \leq \beta \leq |\mathit{lits}(\eta_2)|$, and $0 \leq \sigma \leq |\mathit{lits}(\eta)|$. We use the notational conventions in (4a)–(4b) that $y_{\ell,1} = \ell$ for all leaves ℓ, and that $y_{\eta,0} = 1$ and $y_{\eta,|\mathit{lits}(\eta)|+1} = 0$ for all nodes η (so that clauses containing $y_{\eta,0}$ or $y_{\eta,|\mathit{lits}(\eta)|+1}$ can be simplified to binary clauses or be omitted when they are satisfied). The clauses $C_\eta^{\Rightarrow}(\alpha, \beta, \sigma)$ in (4b) are not necessarily added to the clause database of the MaxSAT solver, but are sometimes included for improved propagation.

We now turn to the question of how to derive the clauses (4a)–(4b) encoding the meaning of the counting variables $y_{I,j}$ in the proof. This is a two-step process. First, reified pseudo-Boolean (and, in general, non-clausal) constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ as in (1a)–(1b), encoding that $y_{\eta,j}$ holds if and only if $\sum_{\ell \in lits(\eta)} \ell \geq j$, are derived by redundance-based strengthening. Then the clauses added to the MaxSAT solver are derived from these pseudo-Boolean constraints. Although we omit the details due to space constraints, it is not hard to show that for any internal node η with children η_1 and η_2, the clauses $C^{\Leftarrow}_{\eta}(\alpha, \beta, \sigma)$ and $C^{\Rightarrow}_{\eta}(\alpha, \beta, \sigma)$ in (4a)–(4b) can be derived from the constraints $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,\sigma})$, $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,\sigma})$, $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta_1,\alpha})$, $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta_1,\alpha})$, $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta_2,\beta})$, and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta_2,\beta})$ by standard cutting planes derivations as in [75]. In particular, the certification of these totalizers can be done incrementally: clauses in the encoding can be derived as the corresponding counter variables are lazily introduced in the OLL algorithm.

This approach is also compatible with structure sharing, where subtrees of a previously constructed totalizer tree can be reused (to avoid doing the same work twice). The only constraints from a subtree rooted at η^* that are needed when generating another totalizer encoding at a higher level are the constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta^*,\sigma})$ and $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta^*,\sigma})$ defining the counter variables in the subtree root η^*.

To decrease the memory usage of the proof checker, it can be useful to *delete* reification constraints from the proof once we know that they will no longer be needed. Without structure sharing, for an internal node η, once all clauses that mention $y_{\eta,j}$ are created, the constraints $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ will not be used anymore and can thus be deleted. On the other hand, structure sharing reuses as many counting variables as possible, even over multiple iterations of weight-aware core extraction. This means that $C^{\Leftarrow}_{\mathrm{reif}}(y_{\eta,j})$ and $C^{\Rightarrow}_{\mathrm{reif}}(y_{\eta,j})$ need to be retained, even after all clauses in the totalizer encoding for all parents of node η have been created.

Objective Reformulation. If counting variables $y_{K,i}$ for $i = 2, \ldots, s_K$ have been introduced for the core K, then the objective reformulation with respect to K is derived with the help of the constraint

$$\sum_{b \in K} b \geq 1 + \sum_{i=2}^{s_K} y_{K,i} \tag{5a}$$

(or

$$\sum_{b \in K} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K \tag{5b}$$

in normalized form). The constraint (5b) can in turn be obtained from the core clause K and the reified constraints $C^{\Rightarrow}_{\mathrm{reif}}(y_{K,j})$. It is clear that this should be possible, since the latter constraints define the variables $y_{K,j}$ precisely so that (5b) should hold, and we refer to Algorithm 5 in [38] for the details. Also, each time a new counting variable $y_{K,j}$ is introduced for a core K, we add it to (5b) to maintain this constraint as an invariant.

To illustrate how this update works, suppose we have a core $K \doteq \sum_{i=1}^{n} b_i \geq 1$ for which $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K-1} \overline{y}_{K,i} \geq s_K - 1$ has already been derived. The next counting variable y_{K,s_K} is introduced by the reification $s_K \cdot \overline{y}_{K,s_K} + \sum_{i=1}^{n} b_i \geq s_K$. The previous constraint is multiplied by $s_K - 1$ and added to the new reified constraint, yielding $s_K \cdot \sum_{i=1}^{n} b + (s_K - 1) \cdot \sum_{i=2}^{s_K-1} \overline{y}_{K,i} + s_K \cdot \overline{y}_{K,s_K} \geq (s_K - 1) \cdot s_K + 1$. Dividing this last constraint by s_K results in $\sum_{i=1}^{n} b + \sum_{i=2}^{s_K} \overline{y}_{K,i} \geq s_K$, which is the desired updated constraint.

For a set of extracted cores $\mathcal{K}$, we can derive the objective reformulation constraint $O_{orig} \geq O_{ref}$ by multiplying (5b) for each $K \in \mathcal{K}$ by the cost $w(K, O_{ref})$ of K and summing up all these multiplied constraints. The fact that we have an inequality $O_{orig} \geq O_{ref}$ rather than an equality is due to the incremental use of totalizers. More specifically, if $s_K = |lits(K)|$ would hold for every $K \in \mathcal{K}$, it would be possible to derive $O_{orig} = O_{ref}$ instead. Here we would like to stress one subtlety for developing proof logging for OLL: as the algorithm progresses and more output variables of totalizers are introduced (i.e., the counters s_K increase), the reformulated objective potentially also increases—because of added counted variables when s_K increases we have the inequality $O_{orig} \geq O_{ref}^{new} \geq O_{ref}^{old}$. For this reason, the old constraint $O_{orig} \geq O_{ref}^{old}$ cannot be used to derive $O_{orig} \geq O_{ref}^{new}$ after objective reformulation. Instead, we have to derive $O_{orig} \geq O_{ref}$ from scratch each time the solver argues with the reformulated objective. For doing this we need to have access to the entire set $\mathcal{K}$ of cores.

Proving Optimality. When the solver has found an optimal solution and established a matching lower bound, optimality is certified in the proof log using a proof by contradiction from the objective reformulation constraint $O_{orig} \geq O_{ref}$ in (2) and the (normalized form of the) objective-improving constraint $O_{orig} \leq UB - 1$ in (3b). If we add these two constraints and cancel like terms, we get

$$\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \cdot \overline{b}' \geq 1 - UB + LB + \sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \,. \qquad (6)$$

Since we have $UB = LB$ when the optimal solution has been found, and since $\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b') \cdot \overline{b}'$ cannot possibly exceed $\sum_{b' \in lits(O_{ref})} coeff(O_{ref}, b')$, the constraint (6) can be simplified to contradiction $0 \geq 1$.

Intrinsic At-Most-One Constraints. Certifying intrinsic at-most-one constraints for a set $\mathcal{S} \subseteq lits(O_{ref})$ of literals requires deriving (i) the at-most-one constraint stating that at most one $b \in \mathcal{S}$ is assigned to 0 by any solution and (ii) constraints defining the variable $\ell_{\mathcal{S}}$. Such sets $\mathcal{S}$ are detected by unit propagation that implicitly derives implications $\overline{b}_i \Rightarrow b_j$ in the form of binary clauses $b_i + b_j \geq 1$ for every pair of variables in $\mathcal{S}$. In the proof log, all these binary clauses can be obtained by RUP steps, after which the at-most-one constraint $\sum_{b \in \mathcal{S}} \overline{b} \leq 1$ (which is $\sum_{b \in \mathcal{S}} b \geq |\mathcal{S}| - 1$ in normalized form) is derived by a standard cutting planes derivation (see, e.g., [24]).

The reified constraints $\ell_{\mathcal{S}} \Leftarrow \sum_{b \in \mathcal{S}} b \geq |S|$ and $\ell_{\mathcal{S}} \Rightarrow \sum_{b \in \mathcal{S}} b \geq |S|$ defining the variable $\ell_{\mathcal{S}}$ (which are $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \overline{b} \geq 1$ and $\overline{\ell}_{\mathcal{S}} + \sum_{b \in \mathcal{S}} b \geq |\mathcal{S}|$, respectively, in

Table 1: Example proof produced by a certified OLL solver.

id	**Pseudo-Boolean constraint**	**Justification**
(1)	$b_1 + x \geq 1$	input
(2)	$b_2 + \overline{x} \geq 1$	input
(3)	$b_3 + b_4 \geq 1$	input
(4)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 6$	log solution α_1
(5)	$b_1 + b_2 \geq 1$	RUP
(6)	$\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$	reification
(7)	$2\overline{y}_{K_1,2} + b_1 + b_2 \geq 2$	reification
(8)	$5b_1 + 5b_2 + 5\overline{y}_{K_1,2} \geq 10$	$(((5) + (7))/2) \cdot 5$
(9)	$\overline{b}_3 + \overline{b}_4 + 5\overline{y}_{K_1,2} \geq 6$	(4) + (8)
(10)	$\overline{y}_{K_1,2} \geq 1$	RUP
(11)	$b_3 + b_4 \geq 1$	RUP
(12)	$\overline{b}_3 + \overline{b}_4 + y_{K_2,2} \geq 1$	reification
(13)	$2\overline{y}_{K_2,2} + b_3 + b_4 \geq 2$	reification
(14)	$b_3 + b_4 + \overline{y}_{K_2,2} \geq 2$	((11) + (13))/2
(15)	$5\overline{b}_1 + 5\overline{b}_2 + \overline{b}_3 + \overline{b}_4 \geq 7$	log solution α_2
(16)	$5b_1 + 5b_2 + b_3 + b_4 + 5\overline{y}_{K_1,2} + \overline{y}_{K_2,2} \geq 12$	(8) + (14)
(17)	$5\overline{y}_{K_1,2} + \overline{y}_{K_2,2} \geq 7$	(15) + (16), $\perp$

normalized form) are derived by redundance-based strengthening. Note that the latter constraint does not exist in the MaxSAT solver, but we need it in the proof in order to derive the objective reformulation for the at-most-one constraint.

Hardening. Formally, hardening corresponds to deriving $\overline{b} \geq 1$ in the proof for some literal $b \in \mathit{lits}(O_{\mathit{ref}})$ for which $\mathit{UB} < \mathit{LB} + \mathit{coeff}(O_{\mathit{ref}}, b)$ holds. Such an inequality $\overline{b} \geq 1$ is implied by RUP if we first derive the constraint (6), since assigning $b = 1$ results in (6) being contradicting.

Upper Bound Estimation. A final technical proof logging detail is that some implementations of the OLL algorithm for MaxSAT—including the Python-based version of CGSS—do not use the actual cost of the solution found by the SAT solver as the upper bound UB when hardening. In order to avoid the overhead in Python of extracting the solution from the SAT solver, an upper bound estimate $\mathit{UB}_{\mathit{est}}$ is computed instead based on the initial assignment passed to the SAT solver in the call. Since any valid estimate is at least the cost of the solution found (i.e., $\mathit{UB}_{\mathit{est}} \geq \mathit{UB}$), hardening steps based on $\mathit{UB}_{\mathit{est}}$ can be justified by first deriving $O_{\mathit{orig}} \leq \mathit{UB}_{\mathit{est}} - 1$, which follows from the latest objective-improving constraint (3a). However, in order to handle solutions correctly in the proof, the proof logging routines need to extract the solution found by the solver and compute the actual cost, which means that a Python-based solver will not be able to avoid this overhead when running with proof logging.

Worked-Out Example. We end this section with a complete, worked-out example of OLL solving and proof logging for the toy MaxSAT instance (F, O) with formula $F = \{(b_1 \vee x), (\neg x \vee b_2), (b_3 \vee b_4)\}$ and objective $O = 5b_1 + 5b_2 + b_3 + b_4$.

After initialization, the internal SAT solver of the OLL algorithm is loaded with the clauses of F and the proof consists of constraints (1)–(3) in Table 1. The OLL search begins by invoking the SAT solver on the clauses in F in order to check the existence of any solutions. Assume the SAT solver returns the solution α_1 assigning $b_1 = b_3 = b_4 = 1$ and $b_2 = x = 0$. This solution has objective value $O(\alpha_1) = O_{orig}(\alpha_1) = 7$ so the algorithm updates $UB = 7$ and logs the objective-improving constraint (4) in Table 1 equivalent to $O_{orig} \leq 6$.

Assume the stratification bound w_{strat} is initialised to 2. Then the solver is invoked with $b_1 = b_2 = 0$ and returns the core $K_1 \doteq b_1 + b_2 \geq 1$, which is added to the proof as constraint (5). As already mentioned, core clauses are guaranteed to be RUP with respect to the set of clauses in the SAT solver database, which are also added to the proof.

For simplicity, we ignore WCE and structure sharing in this example, meaning that the solver next reformulates the objective based on K_1 by introducing clauses enforcing $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ for the new counting variable $y_{K_1,2}$. This is done by (i) introducing the pseudo-Boolean constraints (6) and (7) in Table 1 by reification, and (ii) deriving the clauses corresponding to these constraints. While the MaxSAT solver only uses the implication (6), the proof also requires constraint (7) corresponding to $y_{K_1,2} \Rightarrow (b_1 + b_2 \geq 2)$. Conveniently, in this toy example $y_{K_1,2} \Leftarrow (b_1 + b_2 \geq 2)$ is already the clause $\overline{b}_1 + \overline{b}_2 + y_{K_1,2} \geq 1$, so step (ii) is not needed. For the general case, we derive totalizer clauses as explained in Section 4. Conceptually, we now replace $5b_1 + 5b_2$ by $5y_{K_1,2} + 5$ to obtain the reformulated objective $O_{ref} = b_3 + b_3 + 5y_{K_1,2} + 5$ with lower bound $LB = 5$. The core K_1 says that at least one of b_1 and b_2 must be true, thus incurring a cost of 5, and $y_{K_1,2}$ is added to the objective to indicate if both of them incur cost.

Since it now holds that $\mathit{coeff}(O_{ref}, y_{K_1,2}) + LB = 5 + 5 \geq 7 = UB$, the literal $y_{K_1,2}$ is hardened to 0. In order to certify this hardening step, i.e., derive $\overline{y}_{K_1,2} \geq 1$, the proof logger first derives the objective reformulation constraint $5b_1 + 5b_2 + b_3 + b_4 \geq b_3 + b_4 + 5y_{K_1,2} + 5$ enforced by line (8) in Table 1. The objective-improving and objective reformulation constraints are then added together to get constraint (9), after which $\overline{y}_{K_1,2} \geq 1$ is obtained by a RUP step.

The next SAT solver call with $b_3 = b_4 = 0$ returns as core the input clause $b_3 + b_4 \geq 1$, and reformulation (lines (11)–(13)) yields $O_{ref} = 5y_{K_1,2} + y_{K_2,2} + 6$ with $LB = 6$. Now suppose the SAT solver finds the solution α_2 with $b_2 = b_3 = x = 1$ and all other variables set to 0, resulting in the objective-improving constraint (15). Since $O_{orig}(\alpha_2) = 6 = LB$, the solver terminates and reports α_2 to be optimal. To certify that this is correct, another objective reformulation constraint (16) is derived, after which the contradictory constraint (17) is obtained by adding (15) and (16). This proves that solutions with cost less than 6 do not exist.

5 Experimental Evaluation

To evaluate the proof logging techniques developed in this paper, we have implemented them in the state-of-the-art MaxSAT solver CGSS [22, 47], which uses the OLL algorithm and structure-sharing totalizers. We employed VeriPB [76],

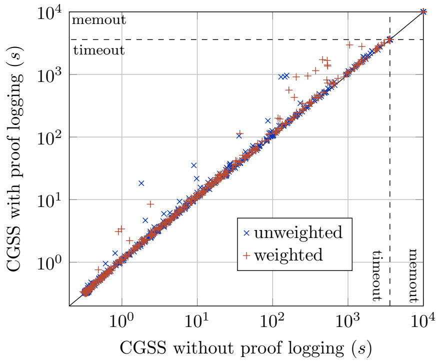

Fig. 1: Running time of CGSS with and without proof logging.

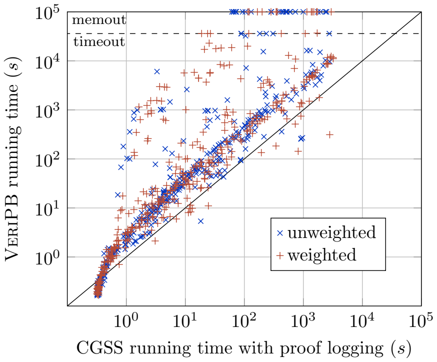

Fig. 2: CGSS running time compared to time required for proof checking.

extended to parse MaxSAT instances in the standard WCNF format, to verify the certificates of correctness emitted by the certifying solver.

Our experiments were conducted on machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory. Each benchmark ran exclusively on a single machine with a memory limit of 14 GB and a time limit of 3 600 seconds for solving with CGSS and 36 000 seconds for checking the certificates with VeriPB. As benchmarks we used all 594 weighted and 607 unweighted instances from the complete track of the MaxSAT Evaluation 2022 [61], where an an instance (F, O) is *unweighted* if all coefficents $\mathit{coeff}(O, \ell)$ are equal. The data from our experiments can be found in [12].

Overhead of Proof Logging. To evaluate the overhead in solver running time, we compared the standard CGSS solver [23] without proof logging (but with the bug fixes discussed below) to CGSS with proof logging as described in this paper. With proof logging 803 instances are solved within the resource limits, which is 3 instances less than without proof logging (see Figure 1). Adding proof logging slowed down CGSS by about 8.8% in the median over all solved instances. For 95% of the instances CGSS with proof logging was at most 36.2% slower. Thus, the proof logging overhead seems perfectly manageable and should present no serious obstacles to using proof logging in core-guided MaxSAT solvers.

Overhead of Proof Checking. To assess the efficiency of proof checking, we compared the running time of CGSS with proof logging to the time taken by VeriPB for checking the generated proofs. The instances that were not solved by CGSS within the resource limits were filtered out, since the running time for checking an incomplete proof is inconclusive.

VeriPB successfully checked the proofs for 747 out of the 803 instances solved by CGSS (see Figure 2); 42 instances failed due to the memory limit and 14 instances failed due to the time limit. Checking the proof took about 3 times

Table 2: Illustration of discovered bug (where $y_{i,k}$ should be read as $y_{K_i,k}$).

#iter	Literals considered ($w_{strat} = 2$)	Core $K_{\#\text{iter}}$ extracted
1	$\{b_i, e_i \mid i = 1 \dots 5\}$	$K_1 = \sum_{i=1}^{5} b_i \geq 1$
2	$\{e_i \mid i = 1 \dots 5\} \cup \{y_{1,2}\}$	$K_2 = y_{1,2} + e_2 + e_4 \geq 1$
3	$\{e_i \mid i = 1 \dots 3, 5\} \cup \{y_{1,2}, y_{1,3}\} \cup \{y_{2,2}\}$	$K_3 = y_{1,3} + e_1 + e_2 + e_5 \geq 1$
4	$\{e_i \mid i = 1 \dots 3\} \cup \{y_{1,2}, y_{1,4}\} \cup \{y_{2,2}, y_{3,2}\}$	$K_4 = y_{1,2} + e_1 + e_2 \geq 1$
5	$\{e_i \mid i = 1 \dots 3\} \cup \{y_{1,4}\} \cup \{y_{2,2}, y_{3,2}, y_{4,2}\}$	$K_5 = e_1 + e_2 + e_3 + y_{1,4} + y_{2,2} \geq 1$
6	$\{e_3\} \cup \{y_{1,5}\} \cup \{y_{2,3}\} \cup \{y_{3,2}, y_{4,2}, y_{5,2}\}$	Result is SAT

#iter	O_{ref} (after reformulation of $K_{\#\text{iter}}$)
0	$10\left(\sum_{i=1}^{5} b_i\right) + 11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + o_1 + o_2$
1	$11e_1 + 14e_2 + 11e_3 + 3e_4 + 2e_5 + 10y_{1,2} + o_1 + o_2 + 10$
2	$11e_1 + 11e_2 + 11e_3 + 2e_5 + 7y_{1,2} + 3y_{1,3} + 3y_{2,2} + o_1 + o_2 + 13$
3	$9e_1 + 9e_2 + 11e_3 + 7y_{1,2} + y_{1,3} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + o_1 + o_2 + 15$
4	$2e_1 + 2e_2 + 11e_3 + \mathbf{8y_{1,3}} + 2y_{1,4} + 3y_{2,2} + 2y_{3,2} + 7y_{4,2} + o_1 + o_2 + 22$
5	$9e_3 + \mathbf{8y_{1,3}} + 2y_{1,5} + y_{2,2} + 2y_{2,3} + 2y_{3,2} + 7y_{4,2} + 2y_{5,2} + o_1 + o_2 + 24$

the solving time in the median for successfully checked instances. About 87% of the successfully checked instances were checked within 10 times the solving time.

Proof checking time compared to solver running time varies widely, but our experiments indicate that the performance of VeriPB is sufficient in most cases, and verification time scales linearly with the size of the proof for a majority of the instances. However, there is room to improve VeriPB, where focus so far has been on proof logging strength rather than performance. For the instances where checking is 100 times slower than solving, the main bottleneck is the proof generated by the SAT solver, which could be addressed by standard techniques for checking *DRAT* proofs, and checking logged solutions (when objective improving constraints (3a) are added) could also be implemented more efficiently.

Bugs Discovered by Proof Logging. Our work on implementing proof logging in CGSS led to the discovery of two bugs, which were also present in the solver RC2 on which CGSS is based, but have now been fixed in CGSS in commit `5526d04` and in RC2 in commit `d0447c3`. The bugs are due to a slightly different implementation of OLL compared to the description in Section 3.

First, when a counting variable $y_{K_{old},i}$ for a core K_{old} appears for the first time in a later core K_{new}, the next counting variable $y_{K_{old},i+1}$ is added to the reformulated objective with coefficient $w\big(K_{new}, O_{new}\big)$ rather than $w\big(K_{old}, O_{old}\big)$. The coefficient of $y_{K_{old},i+1}$ is then further increased when $y_{K_{old},i}$ is found in future cores. Second, rather than computing the upper bound UB from an actual solution, CGSS uses a weaker estimate UB_{est} obtained by summing the current lower bound and the coefficients of all literals b where $\mathit{coeff}(O_{ref}, b) < w_{strat}$ (meaning that these literals were not set to 0 in the SAT solver call, and so could potentially be true in the solution).

The bugs we detected could lead to the solver producing an overly optimistic estimate $UB_{est} < UB$. The first way this can happen is when the contributions of counting variables $y_{K,k}$ in the reformulated objective are underestimated due to too small coefficients. The second bug is when the coefficient of $y_{K_{old},i+1}$ is first lowered below w_{strat} and then raised above this threshold again when $y_{K_{old},i}$ is found in a core. Then CGSS fails to assume $y_{K_{old},i+1} = 0$ in future solver calls. These bugs can result in erroneous hardening as detailed in the next example.

Example 1. Given a MaxSAT instance (F, O) with $F = \left\{\left(\bigvee_{i=1}^{5} b_i\right), (o_1 \vee o_2)\right\} \cup \{b_i \vee e_i \mid i = 1, \ldots, 5\}$ and $O = \left(\sum_{i=1}^{5} 10 \cdot b_i\right) + 11 \cdot e_1 + 14 \cdot e_2 + 11 \cdot e_3 + 3 \cdot e_4 + 2 \cdot e_5 + o_1 + o_2$, assume the stratification bound is $w_{strat} = 2$. Table 2 displays a possible CGSS run for this instance, except that for simplicity we assume one core extraction per iteration and no use of any other heuristics. The upper half of the table lists the variables set to 0 in solver calls, the extracted core, and the lower bound derived from it. The lower half of the table provides the reformulated objective. Even though the coefficient of $y_{K_1,3}$ is increased to 8 after the fourth core, this variable is not set to 0 in subsequent iterations, which allows the solver to finish the stratification level after extracting 6 cores with a solution that sets to true the variables $b_1, b_2, b_3, b_5, e_4, o_1, o_2, y_{K_2,2}$ and $y_{K_1,i}$ for $i = 1, \ldots, 4$, and all other variables to false. The cost of this solution is 45.

Now CGSS would incorrectly estimate $UB_{est} = LB + 4 = 28$, since $y_{K_1,3}$ and $y_{K_2,2}$ (abbreviated as $y_{1,3}$ and $y_{2,2}$ in the table) both have coefficient 1 in the current reformulated objective. This is lower than the cost 45 of the solution found (and even than the optimum 36), and erroneously allows hardening—which considers $y_{K_1,3}$ with the correct coefficient 8—to fix $y_{K_1,3} = 0$, even though b_1, b_2 and b_3 (and hence also $y_{K_1,3}$) are true in every minimal-cost solution.

In our computational experiments there were cases of faulty hardening, but all incorrectly fixed values happened to agree with some optimal solution and so we never observed incorrect results. Proof logging detected the problem, however, since the derivations of the buggy hardening steps failed during proof checking. Interestingly, what proof logging did *not* turn up was any examples of mistaken claims $O_{orig} \leq UB_{est} - 1$ when the cost of a found solution was estimated. The issue with mistaken estimates due to faulty stratification was instead discovered while analyzing and fixing the hardening bug. The moral of this is that even if all results are certified as correct, this does not certify that the code is free from bugs that have not yet manifested themselves. However, proof logging still guarantees that even if the solver would have undiscovered bugs, we can always trust computed results for which the accompanying proofs pass verification.

6 Concluding Remarks

In this work, we develop pseudo-Boolean proof logging techniques for core-guided MaxSAT solving and implement them in the solver CGSS [47] with support for the full range of sophisticated reasoning techniques it uses. To the best of our knowledge, this is the first time a state-of-the-art MaxSAT solver has been enhanced to output machine-verifiable proofs of correctness. We have made a thorough evaluation on benchmarks from the MaxSAT Evaluation 2022 using the

VeriPB proof checker [17, 42], and find that proof logging overhead is perfectly manageable and that proof verification time, while leaving room for improvement, is definitely practically feasible. Our work also showcases the benefit of proof logging as a debugging tool—erroneous proofs produced by CGSS revealed two subtle bugs in the solver that previous extensive testing had failed to uncover.

Regarding proof verification time, further investigation is needed into the rare cases where verification is much slower (say, more than a factor 10) than solving. There are reasons to believe, though, that this is not a problem of MaxSAT proof logging per se, but rather is explained by features not yet added to VeriPB, which is a tool currently undergoing very active development. So far, the proof checker has been optimized for other types of reasoning than the clausal reverse unit propagation (RUP) steps that dominate SAT proofs. Also, VeriPB lacks the ability to trim proofs during checking as in [44]. Finally, introducing a binary proof format in addition to plain-text proofs would be another way to boost performance of proof checking. But these are matters of engineering rather than research, and can be taken care of once the proof logging technology as such has been developed and has proven its worth.

The focus of this work is on core-guided MaxSAT solving, but we would like to extend our techniques to solvers using linear SAT-UNSAT (LSU) solving (such as Pacose [68]) and implicit hitting set (IHS) search (such as MaxHS [28, 29]). Although there are certainly nontrivial technical challenges that will need to be overcome, we are optimistic that our work paves the way towards a unified proof logging system for the full range of modern MaxSAT solving approaches. Going beyond MaxSAT, it would also be interesting to extend VeriPB proof logging to pseudo-Boolean solvers using core-guided search [30] or IHS [73, 74], and perhaps even to similar techniques in constraint programming [36] and answer set programming [5].

Acknowledgements This work was partly carried out while some of the authors were visiting the Simons Institute for the Theory of Computing at UC Berkeley for the extended reunion of the program "Satisfiability: Theory, Practice, and Beyond" during the spring of 2023. We also benefited greatly from the Dagstuhl Seminar 22411 "Theory and Practice of SAT and Combinatorial Solving." Additionally, we acknowledge several inspirational discussions on certifying solvers and proof logging with, among others, Ambros Gleixner, Stephan Gocht, and Ciaran McCreesh. The computational experiments were enabled by resources provided by LUNARC at Lund University.

Jeremias Berg was fully supported by the Academy of Finland under grant 342145. Bart Bogaerts and Dieter Vandesande were supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N) and by the EU ICT-48 2020 project TAILOR (GA 952215). Jakob Nordström was supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: Jünger, M., Reinelt, G. (eds.) Facets of Combinatorial Optimization, pp. 449–481. Springer (2013)
2. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Metamorphic testing of constraint solvers. In: Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18). Lecture Notes in Computer Science, vol. 11008, pp. 727–736. Springer (Aug 2018)
3. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C., Schweitzer, P.: An introduction to certifying algorithms. it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik **53**(6), 287–293 (Dec 2011)
4. Alviano, M., Dodaro, C., Ricca, F.: A MaxSAT algorithm using cardinality constraints of bounded size. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI '15). pp. 2677–2683. AAAI Press (2015)
5. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP '12). Leibniz International Proceedings in Informatics (LIPIcs), vol. 17, pp. 211–221 (Sep 2012)
6. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12). Lecture Notes in Computer Science, vol. 7514, pp. 86–101. Springer (Oct 2012)
7. Ansótegui, C., Gabàs, J.: WPM3: An (in)complete algorithm for weighted partial MaxSAT. Artificial Intelligence **250**, 37–57 (2017)
8. Bacchus, F., Järvisalo, M., Martins, R.: Maximum satisfiabiliy. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 929–991. IOS Press, 2nd edn. (Feb 2021)
9. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of Boolean cardinality constraints. In: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03). Lecture Notes in Computer Science, vol. 2833, pp. 108–122. Springer (Sep 2003)
10. Barth, P.: A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik (Jan 1995)
11. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97). pp. 203–208 (Jul 1997)
12. Berg, J., Bogaerts, B., Nordström, J., Oertel, A., Vandesande, D.: Experimental repository for "Certified core-guided MaxSAT solving". https://doi.org/10.5281/zenodo.7709687 (May 2023)
13. Berg, J., Järvisalo, M.: Weight-aware core extraction in SAT-based MaxSAT solving. In: Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP '17). Lecture Notes in Computer Science, vol. 10416, pp. 652–670. Springer (Aug 2017)
14. Biere, A.: Tracecheck. http://fmv.jku.at/tracecheck/ (2006)
15. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press, 2nd edn. (Feb 2021)

16. Bixby, R., Rothberg, E.: Progress in computational mixed integer programming—A look back from the other side of the tipping point. Annals of Operations Research **149**(1), 37–41 (Feb 2007)
17. Bogaerts, B., Gocht, S., McCreesh, C., Nordström, J.: Certified symmetry and dominance breaking for combinatorial optimisation. In: Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22). pp. 3698–3707 (Feb 2022)
18. Bogaerts, B., McCreesh, C., Nordström, J.: Solving with provably correct results: Beyond satisfiability, and towards constraint programming (Aug 2022), tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`
19. Bonet, M.L., Levy, J., Manyà, F.: Resolution for Max-SAT. Artificial Intelligence **171**(8-9), 606–618 (2007)
20. Buss, S.R., Nordström, J.: Proof complexity and SAT solving. In: Biere et al. [15], chap. 7, pp. 233–350
21. Calabro, C., Impagliazzo, R., Paturi, R.: The complexity of satisfiability of small depth circuits. In: Revised Selected Papers from the 4th International Workshop on Parameterized and Exact Computation (IWPEC '09). Lecture Notes in Computer Science, vol. 5917, pp. 75–85. Springer (Sep 2009)
22. Certifying version of the CGSS core-guided MaxSAT solver with structure sharing. `https://gitlab.com/MIAOresearch/software/certified-cgss`
23. CGSS, a core guided Max-SAT-algorithm using structure sharing technique for enhanced cardinality constraints, built on RC2 and PySAT. `https://bitbucket.org/coreo-group/cgss/`
24. Cook, W., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discrete Applied Mathematics **18**(1), 25–38 (Nov 1987)
25. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Mathematical Programming Computation **5**(3), 305–344 (Sep 2013)
26. Cruz-Filipe, L., Heule, M.J.H., Hunt Jr., W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: Proceedings of the 26th International Conference on Automated Deduction (CADE-26). Lecture Notes in Computer Science, vol. 10395, pp. 220–236. Springer (Aug 2017)
27. Cruz-Filipe, L., Marques-Silva, J.P., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '17). Lecture Notes in Computer Science, vol. 10205, pp. 118–135. Springer (Apr 2017)
28. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13). Lecture Notes in Computer Science, vol. 7962, pp. 166–181. Springer (Jul 2013)
29. Davies, J., Bacchus, F.: Postponing optimization to speed up MAXSAT solving. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP '13). Lecture Notes in Computer Science, vol. 8124, pp. 247–262. Springer (2013)
30. Devriendt, J., Gocht, S., Demirović, E., Nordström, J., Stuckey, P.: Cutting to the core of pseudo-Boolean optimization: Combining core-guided search with cutting planes reasoning. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21). pp. 3750–3758 (Feb 2021)
31. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4), 1–26 (Mar 2006)

32. Eifler, L., Gleixner, A.: A computational status update for exact rational mixed integer programming. In: Proceedings of the 22nd International Conference on Integer Programming and Combinatorial Optimization (IPCO '21). Lecture Notes in Computer Science, vol. 12707, pp. 163–177. Springer (May 2021)
33. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using pseudo-Boolean reasoning. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20). pp. 1486–1494 (Feb 2020)
34. Filmus, Y., Mahajan, M., Sood, G., Vinyals, M.: MaxSAT resolution and subcube sums. In: Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20). Lecture Notes in Computer Science, vol. 12178, pp. 295–311. Springer (Jul 2020)
35. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06). Lecture Notes in Computer Science, vol. 4121, pp. 252–265. Springer (Aug 2006)
36. Gange, G., Berg, J., Demirović, E., Stuckey, P.J.: Core-guided and core-boosted search for constraint programming. In: Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '20). Lecture Notes in Computer Science, vol. 12296, pp. 205–221. Springer (Sep 2020)
37. Gillard, X., Schaus, P., Deville, Y.: SolverCheck: Declarative testing of constraints. In: Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19). Lecture Notes in Computer Science, vol. 11802, pp. 565–582. Springer (Oct 2019)
38. Gocht, S., Martins, R., Nordström, J., Oertel, A.: Certified CNF translations for pseudo-Boolean solving. In: Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 16:1–16:25 (Aug 2022)
39. Gocht, S., McBride, R., McCreesh, C., Nordström, J., Prosser, P., Trimble, J.: Certifying solvers for clique and maximum common (connected) subgraph problems. In: Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20). Lecture Notes in Computer Science, vol. 12333, pp. 338–357. Springer (Sep 2020)
40. Gocht, S., McCreesh, C., Nordström, J.: Subgraph isomorphism meets cutting planes: Solving with certified solutions. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20). pp. 1134–1140 (Jul 2020)
41. Gocht, S., McCreesh, C., Nordström, J.: An auditable constraint programming solver. In: Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22). Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, pp. 25:1–25:18 (Aug 2022)
42. Gocht, S., Nordström, J.: Certifying parity reasoning efficiently using pseudo-Boolean proofs. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21). pp. 3768–3777 (Feb 2021)
43. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03). pp. 886–891 (Mar 2003)
44. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13). pp. 181–188 (Oct 2013)

45. Heule, M.J.H., Hunt Jr., W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Proceedings of the 24th International Conference on Automated Deduction (CADE-24). Lecture Notes in Computer Science, vol. 7898, pp. 345–359. Springer (Jun 2013)
46. Ignatiev, A., Morgado, A., Marques-Silva, J.P.: RC2: an efficient MaxSAT solver. Journal on Satisfiability, Boolean Modeling and Computation **11**(1), 53–64 (Sep 2019)
47. Ihalainen, H., Berg, J., Järvisalo, M.: Refined core relaxation for core-guided MaxSAT solving. In: 27th International Conference on Principles and Practice of Constraint Programming (CP '21). Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, pp. 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
48. Ihalainen, H., Berg, J., Järvisalo, M.: Clause redundancy and preprocessing in maximum satisfiability. In: Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22). Lecture Notes in Computer Science, vol. 13385, pp. 75–94. Springer (Aug 2022)
49. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. Journal of Computer and System Sciences **62**(2), 367–375 (Mar 2001), preliminary version in *CCC '99*
50. Karpinski, M., Piotrów, M.: Competitive sorter-based encoding of PB-constraints into SAT. In: Proceedings of Pragmatics of SAT. EPiC Series in Computing, vol. 59, pp. 65–78. EasyChair (2018)
51. Karpinski, M., Piotrów, M.: Encoding cardinality constraints using multiway merge selection networks. Constraints **24**(3-4), 234–251 (2019)
52. Kraiczy, S., McCreesh, C.: Solving graph homomorphism and subgraph isomorphism problems faster through clique neighbourhood constraints. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI '21). pp. 1396–1402 (Aug 2021)
53. Larrosa, J., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A framework for certified Boolean branch-and-bound optimization. Journal of Automated Reasoning **46**(1), 81–102 (2011)
54. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation **7**, 59–64 (Jul 2010)
55. Leivo, M., Berg, J., Järvisalo, M.: Preprocessing in incomplete maxsat solving. In: Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20). Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 347–354. IOS Press (2020)
56. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 336, pp. 903–927. IOS Press (2021)
57. Li, C., Xu, Z., Coll, J., Manyà, F., Habet, D., He, K.: Boosting branch-and-bound MaxSAT solvers with clause learning. AI Communications **35**(2), 131–151 (2022)
58. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Annals of Mathematics and Artificial Intelligence **62**(3-4), 317–343 (2011)
59. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5), 506–521 (May 1999), preliminary version in *ICCAD '96*
60. Martins, R., Joshi, S., Manquinho, V.M., Lynce, I.: Incremental cardinality constraints for MaxSAT. In: Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14). Lecture Notes in Computer Science, vol. 8656, pp. 531–548. Springer (Sep 2014)

61. MaxSAT evaluation 2022. https://maxsat-evaluations.github.io/2022 (Aug 2022)
62. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (May 2011)
63. Morgado, A., Dodaro, C., Marques-Silva, J.P.: Core-guided MaxSAT with soft cardinality constraints. In: Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14). Lecture Notes in Computer Science, vol. 8656, pp. 564–573. Springer (Sep 2014)
64. Morgado, A., Ignatiev, A., Bonet, M.L., Marques-Silva, J.P., Buss, S.R.: DRMaxSAT with MaxHS: First contact. In: Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19). Lecture Notes in Computer Science, vol. 11628, pp. 239–249. Springer (Jul 2019)
65. Morgado, A., Marques-Silva, J.: On validating Boolean optimizers. In: Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI '11). pp. 924–926 (2011)
66. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC '01). pp. 530–535 (Jun 2001)
67. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided maxsat resolution. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI '14). pp. 2717–2723. AAAI Press (2014)
68. Paxian, T., Reimer, S., Becker, B.: Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In: Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18). Lecture Notes in Computer Science, vol. 10929, pp. 37–53. Springer (Jul 2018)
69. Py, M., Cherif, M.S., Habet, D.: Towards bridging the gap between SAT and Max-SAT refutations. In: Proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '20). pp. 137–144 (Nov 2020)
70. Py, M., Cherif, M.S., Habet, D.: A proof builder for Max-SAT. In: Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21). Lecture Notes in Computer Science, vol. 12831, pp. 488–498. Springer (Jul 2021)
71. Py, M., Cherif, M.S., Habet, D.: Proofs and certificates for Max-SAT. Journal of Artificial Intelligence Research **75**, 1373–1400 (Dec 2022)
72. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006)
73. Smirnov, P., Berg, J., Järvisalo, M.: Improvements to the implicit hitting set approach to pseudo-Boolean optimization. In: Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 13:1–13:18 (Aug 2022)
74. Smirnov, P., Berg, J., Järvisalo, M.: Pseudo-Boolean optimization by implicit hitting sets. In: Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21). Leibniz International Proceedings in Informatics (LIPIcs), vol. 210, pp. 51:1–51:20 (Oct 2021)
75. Vandesande, D., De Wulf, W., Bogaerts, B.: QMaxSATpb: A certified MaxSAT solver. In: Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22). Lecture Notes in Computer Science, vol. 13416, pp. 429–442. Springer (Sep 2022)
76. VeriPB: Verifier for pseudo-Boolean proofs. https://gitlab.com/MIAOresearch/software/VeriPB