Trimming Pseudo-Boolean Proofs

Berhan Oumer Adame
Vrije Universiteit Brussel, Belgium
Arba Minch University, Ethiopia
berhan.oumer.adame@vub.be

Bart Bogaerts
KU Leuven, Belgium
Vrije Universiteit Brussel, Belgium
bart.bogaerts@kuleuven.be

Benjamin Bogø
University of Copenhagen, Denmark
Lund University, Sweden
bebo@di.ku.dk

Simon Dold
University of Basel, Switzerland
simon.dold@unibas.ch

Arthur Gontier
University of Glasgow, United Kingdom
arthur.gontier@glasgow.ac.uk

Wietze Koops
Lund University, Sweden
University of Copenhagen, Denmark
wietze.koops@cs.lth.se

Ciaran McCreesh
University of Glasgow, United Kingdom
ciaran.mccreesh@glasgow.ac.uk

Matthew J. McIlree
University of Glasgow, United Kingdom
m.mcilree.1@research.gla.ac.uk

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden
jn@di.ku.dk

Andy Oertel
Lund University, Sweden
University of Copenhagen, Denmark
andy.oertel@cs.lth.se

Adrian Rebola-Pardo
Vienna University of Technology, Austria
Johannes Kepler University Linz, Austria
adrian.rebola@tuwien.ac.at

Mark Turnbull
University of Glasgow, United Kingdom
mark.turnbull@glasgow.ac.uk

***Abstract*—Modern combinatorial solvers are very efficient but also highly complex and therefore error-prone. The most successful approach to ensure correctness—*proof logging*—is now the accepted standard for Boolean satisfiability (SAT) solving, where *proof trimming* is an additional crucial technique to reduce proof size and proof checking time. However, efficient proof logging has remained a challenge for richer combinatorial paradigms, and trimming has not been supported for the more complex proof logging systems used. In this work, we present a proof trimmer covering the full VERIPB proof format, bringing the advantages of trimming to a wide range of paradigms such as MaxSAT, pseudo-Boolean optimisation, subgraph solving, and constraint programming. Our experimental evaluation demonstrates that including proof trimming in the pipeline leads to overall reductions of proof size by a factor of 6.37 and of total proof checking time (including formally verified checking with CAKEPB) by a factor of 1.81.**

The first author received funding from VLIR-UOS and DGD (AMU-IUC ET2022IUC035A101). The second author was funded by Fonds Wetenschappelijk Onderzoek — Vlaanderen (project G064925N) and 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. The fourth author was funded by the Swiss National Science Foundation (SNSF) as part of the project "Unifying the Theory and Algorithms of Factored State-Space Search" (UTA). The fifth and seventh authors were supported by the Engineering and Physical Sciences Research Council grant number EP/X030032/1. The seventh author was also supported by a Royal Academy of Engineering research fellowship and, together with the eighth author, by the Advanced Research and Invention Agency. The sixth and tenth authors were supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The ninth author was supported by the Independent Research Fund Denmark grant 9040-00389B and the Swedish Research Council grant 2024-05801. The eleventh author was supported by FWF grant 10.55776/COE12 and WWTF grant 10.47379/ICT25051.

I. INTRODUCTION

Over the last decades, combinatorial optimisation has become an increasingly important tool both in industry and scientific research, where it is used, e.g., to control factories [33], [59], [74], manage workforce allocation and scheduling [15], [70], direct air traffic [4], support hardware and software development [2], [13], [14], design medical treatment [60], and even to prove theorems in pure mathematics [48], [53], [84]. These applications draw on a range of combinatorial solving paradigms such as Boolean satisfiability (SAT) solving, SAT-based optimisation (MaxSAT), pseudo-Boolean optimisation, graph solving, constraint programming (CP), and mixed-integer programming (MIP). Although performance improvements in combinatorial solving since the turn of the millennium have been nothing short of astonishing, this has come at the cost of a steep increase in the complexity of the software tools. As a result, even mature solvers are known to be buggy, sometimes erroneously claiming infeasibility or optimality, or returning "solutions" that fail to satisfy all constraints [3], [21], [35], [36], [64], [86].

The SAT community has tackled this problem very successfully by making solvers *certifying* [62], meaning that they use *proof logging* to output machine-verifiable certificates that their answers are valid. Two key concerns in this process are *efficiency* and *correctness*. In order to be practically useful, proof logging should incur minimal overhead. In DRAT [46], [47], [91], the *de facto* standard proof format for SAT solving, this is achieved by making the proof consist simply of the sequence of clausal constraints inferred by the solver, without

https://doi.org/

details of how precisely they were obtained. However, this makes the task of validating the proof more difficult. In order to achieve formally verified results, proof checking is therefore typically performed in two passes:

- First, an efficient but unverified proof checker *elaborates* the proof, filling in all details on how the clauses can be derived so that no search is needed.
- Second, a formally verified proof checker goes over the more explicit, elaborated, proof and validates all proof steps.

In this way, SAT proof checking provides extremely strong correctness guarantees [23], [24], [57], [85]. An important way of increasing efficiency further is to elaborate only the part of the proof that is actually needed to reach the final verdict (which for SAT proof logging is that the formula is unsatisfiable) and trim away the rest of the proof [46]. At a high level, the checker does so by starting at the end of the proof, marking the constraints that were used in the final step to derive the verdict. The checker then reads the proof backwards. Any unmarked constraint has not been used to reach the final conclusion and can be removed, but marked constraints need to be kept. Moreover, for each marked constraint, the checker needs to determine a set of previous constraints that can be used to derive it, recursively marking these as well. This process yields a (usually much) shorter *trimmed* proof, speeding up the formally verified proof checking (though in order to be efficient, trimming is a heuristic procedure, and so there are no guarantees that the trimmed proof is minimal).

In addition to improving the performance of proof checking, reducing proof size can be beneficial when proofs are to be stored for auditing at a later date [8], [78]. Proof logging combined with trimming also provides new possibilities for in-depth scientific analysis of algorithms [55], [80], [83] by helping to identify which inference techniques and heuristics contribute to solver success on particular instances. Furthermore, trimming can extract smaller unsatisfiable subformulas of a formula, so-called *UNSAT cores* [9], [26], and also generate smaller explanations for why a solution is optimal [16].

Progress on proof logging in settings beyond SAT solving has been much more limited, but this has changed in recent years with the introduction of *pseudo-Boolean proof logging* with VERIPB [6], [17], [37], [43], [66] using 0–1 integer linear inequalities. Somewhat surprisingly, this proof language has turned out to provide a uniform method of certifying correctness for a broad range of combinatorial solving paradigms including MaxSAT [10], [11], [49], [88], [89], pseudo-Boolean solving and optimisation [38], [50], [56], graph solving [30], [39]–[41], constraint programming [31], [42], [67]–[69], and automated planning [29], and to also support advanced SAT reasoning techniques [6], [17], [43] and dynamic programming and decision diagrams [25] as well as settings such as model enumeration [63] and multi-objective optimisation [51], [52]. However, proof logging and checking overheads have generally been unacceptably large. In particular, proper tools for *proof trimming* have been missing—while some work has been done in the limited setting of *implicational* pseudo-Boolean proofs [12], [71], there has been no support for the full range of stronger reasoning rules in pseudo-Boolean proof logging (such as the strengthening rules used for without-loss-of-generality arguments) that make the proof trimming task highly nontrivial.

In this work, we design, implement, and evaluate proof trimming methods for VERIPB, which immediately makes proof trimming available for all the combinatorial solving paradigms just listed, not only to achieve faster proof checking but also for other applications as explained above.

The fact that VERIPB proof logging is used by many different types of combinatorial solvers makes it quite challenging to design a general-purpose trimmer. The proof rules allow constraints to be derived by a mix of explicit and implicit derivations. Some rules, such as *cutting planes* derivations [22] in *reverse Polish notation* derive constraints by listing an explicit sequence of syntactic operations on previously derived constraints, but do not specify what the derived constraint looks like (since this is clear from the operations performed, and since omitting the description of the final constraint significantly reduces proof size). However, just as for the DRAT proof format there are also more implicit rules such as *reverse unit propagation (RUP)* [31], [44], [87] for claiming "obvious" constraints without justification. In order for the proof checker to be able to reconstruct explicit derivations of such constraints inferred by implicit rules, it needs to know exactly which constraints are currently in the database. But if the proof is being read backwards while trimming, it is not possible to infer the syntactic form of constraints derived by reverse Polish notation lines. Note that this problem does not occur in SAT proof formats such as DRAT and LRAT, where all derived constraints are listed explicitly.

An even trickier case is reasoning by *strengthening* as mentioned above, which allows adding a non-implied constraint C to a formula $\mathcal{F}$ through a without-loss-of-generality argument. For instance, if $\mathcal{F}$ encodes a graph colouring problem, then strengthening can add a constraint C forcing, without loss of generality, some specific vertex v to be coloured red. Clearly some care is needed here: once we have derived C, we cannot repeat the same argument to derive that v can also be coloured green without loss of generality. To maintain soundness, strengthening steps in VERIPB induce a *proof obligation* for every currently available constraint C' (from the formula $\mathcal{F}$ or previously inferred and not deleted). To fulfil this proof obligation, it is necessary to show that inferring C is consistent with satisfying C' as well. These considerations lead to several new challenges.

First, in order to parse and check proof logging rules backwards, the full context (consisting of the constraints derived so far) needs to be known. Our solution is to introduce a *decoration* phase, which is a lightweight forward pass that adds sufficient information to the proof to enable backward trimming. However, decoration cannot be done too eagerly, because some solvers will write lots of proof steps with an explicit derivation of a large constraint immediately followed

by an implicit simplification to a much smaller constraint. We therefore need to decorate lazily in order for the intermediate constraints not to blow up the proof size.

A second challenge is related to the phenomenon of *interference* [45], namely that non-implicational inference rules such as strengthening behave non-monotonically. Each previously derived constraint still in the database can potentially interfere with the application of a strengthening step in that it imposes a proof obligation. However, during a backward trimming pass over the proof, the checker does not yet know which constraints will ultimately be included in the final trimmed proof. It is therefore not possible to know which proof obligations must be checked. A potential solution is to check proof obligations for marked constraints, and then mark all constraints used for showing these obligations, repeating this process until a fixpoint is reached [75]. The idea would be that all constraints not marked by this fixpoint process could safely be deleted right before the derivation of C without affecting any parts of the proof that come later. However, the problem turns out to be more complex, as discussed further below, and implementing a correct solution is far from trivial.

Deletion steps are another source of difficulty. In SAT proof formats such as DRAT and LRAT, erasing previously derived constraints is always possible without restriction, since such proofs are only used to establish unsatisfiability. However, VERIPB proofs can also be used for optimisation problems with upper and lower bounds, and even for satisfiable instances, and in such settings deletion steps have to be carefully managed. In addition, VERIPB has stronger proof rules (such as the *dominance-based strengthening* rule [17] and *strengthening-to-core* mode [11]) that impose additional restrictions on deletions. This means that when trimming proofs, deletion steps cannot be moved around or inserted arbitrarily. To overcome all of these technical challenges, we introduce a novel proof trimming technique with *conditional markings*. This means that we keep more fine-grained control on which constraints need to be included in the final trimmed proof based on dependencies on other constraints, where these other constraints might or might not become marked later during the backward trimming pass.

Our proof trimming algorithm consists of three passes. At a high level, the first *decoration* pass adds information that is implicit in the proof but needs to be made explicit. In the second *backward trimming* pass, we mark constraints (conditionally and unconditionally) to identify which parts must be retained or can be trimmed away. A third and final *compaction* pass performs the actual trimming of constraints and prints the trimmed proof, eliminating book-keeping details that are no longer needed.

We have implemented our trimming method on top of the most recent version of VERIPB [90] and have run extensive experiments on SAT solving with symmetry breaking, MaxSAT solving, pseudo-Boolean optimisation, subgraph solving, and constraint programming. Algorithms in these different combinatorial solving paradigms can produce proofs with widely differing properties, and our results vary for different types of problems. However, overall we see significant reductions across these applications for proof size and proof checking time (including for the formally verified backend CAKEPB), which shows that our proof trimming method is viable for pseudo-Boolean proof logging in general.

The rest of this paper is organised as follows. After describing the VERIPB proof system in Section II, we explain our trimming method in Section III. Section IV delves into conditional markings, and Section V discusses other design decisions and implementation details. Experimental results are presented in Section VI, after which we conclude in Section VII.

II. PRELIMINARIES

We start by reviewing pseudo-Boolean reasoning and the VERIPB system, referring to [6], [17], [18] for more details.

We use x to denote Boolean ($\{0,1\}$-valued) variables and write $\overline{x} = 1 - x$ for negation. A *literal* ℓ is a variable or its negation (where we also define $\overline{\overline{x}} = x$ for convenience). A *pseudo-Boolean (PB) constraint* C is a 0–1 integer linear inequality $\sum_i a_i \ell_i \geq A$, which we can assume without loss of generality to be in *normalised* form with all literals over distinct variables and coefficients a_i and right-hand side A non-negative. A *pseudo-Boolean formula* is a conjunction or a set of PB constraints, whichever view is more convenient. The *negation* $\neg C$ of a constraint $C = \sum_i a_i \ell_i \geq A$ written in normalised form is $\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$.

A *(partial) assignment* α is a (partial) function mapping variables to $\{0,1\}$. We extend assignments to literals by $\alpha(\overline{x}) = 1 - \alpha(x)$, respecting the meaning of negation, and sometimes write $\alpha(\ell) = *$ when ℓ is not in the domain of α. For a constraint $C = \sum_i a_i \ell_i \geq A$, we write $C{\restriction_\alpha}$ for the constraint obtained from C by substituting literals ℓ in the domain of α by $\alpha(\ell)$ and simplifying. The constraint C is *satisfied* by α if $\sum_{\alpha(\ell_i)=1} a_i \geq A$ (assuming normalised form) and a formula $\mathcal{F}$ is satisfied if all its constraints are, in which case α is a *solution* to $\mathcal{F}$. We say that $\mathcal{F}$ *implies* C, denoted by $\mathcal{F} \models C$, if all assignments that satisfy $\mathcal{F}$ also satisfy C, and for formulas we write $\mathcal{F} \models \mathcal{F}'$ if $\mathcal{F} \models C$ for all $C \in \mathcal{F}'$. The *slack* of $C = \sum_i a_i \ell_i \geq A$ under α, defined as $slack(C;\alpha) = \sum_{\alpha(\ell_i)\in\{1,*\}} a_i - A$, is non-negative if and only if C is satisfied by α or some extension of α. The constraint C is *falsified* by α if $slack(C;\alpha) < 0$. Otherwise, if $slack(C;\alpha) < a_i$ for some term $a_i\ell_i$ in C, and $\alpha(\ell_i) = *$, we say that C *unit propagates* ℓ_i to true under α (since falsifying ℓ_i would also falsify C). *Substitutions* ω map variables to $\{0,1\}$ or literals; they are extended to literals by respecting negation, and $C{\restriction_\omega}$ denotes C with literals ℓ in the domain of ω replaced by $\omega(\ell)$.

A VERIPB proof for the decision problem for a pseudo-Boolean formula $\mathcal{F}$ maintains a *proof configuration* $(\mathcal{C}, \mathcal{D})$, consisting of a pair of sets of *core constraints* $\mathcal{C}$ and *derived constraints* $\mathcal{D}$.[1] The configuration is initialised to $(\mathcal{F}, \emptyset)$ with

[1]For simplicity of exposition, we ignore here optimisation problems and so-called *dominance-based strengthening*, since that would require a more elaborate proof configuration. However, the trimming techniques presented in Sections III and IV are fully sufficient to also deal with objective functions and dominance, and our trimmer is able to do so as discussed in Section V-B.

the input formula in the core set and the derived set empty. The proof system rules are used to modify the core and derived sets by adding and removing constraints, with new constraints receiving consecutive positive integers as *constraint IDs*, until the proof is concluded by either a solution or a configuration containing contradiction $0 \geq 1$.

VERIPB proofs can switch between *regular* and *strengthening-to-core* mode, with the latter maintaining the invariant $\mathcal{C} \models \mathcal{D}$ that all constraints in the derived set $\mathcal{D}$ are implied by the core set $\mathcal{C}$ (which makes the effect of the proof rules slightly different between the two modes). Below, we try to describe enough of the VERIPB rules to provide an understanding of what tasks need to be solved during proof trimming.

Implied constraints. Constraints C semantically implied by $\mathcal{C} \cup \mathcal{D}$ can be added to $\mathcal{D}$ using the following two rules:

`pol`. Provide a *cutting planes* derivation [22] of C from $\mathcal{C} \cup \mathcal{D}$ in reverse Polish notation by applying scalar multiplication, division, addition, and similar syntactic rules on constraints identified by their constraint IDs.

`rup`. Ask the checker to verify $\mathcal{C} \cup \mathcal{D} \models C$ by performing *reverse unit propagation (RUP)* [31], unit propagating iteratively on $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$ and extending the truth value assignment with all propagated literals until a *conflict* is reached in the form of a falsified constraint.

Switching mode. Turning off strengthening-to-core mode is always allowed, but turning it on requires $\mathcal{D}$ to be empty.

Moving constraints. Constraints can be moved from the derived set $\mathcal{D}$ to the core set $\mathcal{C}$, but not vice versa.

Redundance-based strengthening. A non-implied constraint C can be inferred by explicitly specifying a *witness* substitution ω together with a proof of the implication $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \models (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega$ [19], [43]. The constraints in $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega$ are called *proof obligations* or *proof goals*, and require implicational *subproofs* composed of, e.g., `pol` or `rup` steps, or should otherwise follow by simple steps that can be automatically inferred by the checker.

- In regular mode, C is then added to $\mathcal{D}$.
- In strengthening-to-core mode, C is added to $\mathcal{C}$; in this case, proving $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \models (\mathcal{C} \cup \{C\})\restriction_\omega$ is sufficient, since $\mathcal{D}\restriction_\omega$ is then implied by $\mathcal{C}\restriction_\omega$.

Example II.1. To demonstrate the power of the strengthening rule, let us illustrate how it can be used to break symmetries. For example, for $\mathcal{C} \cup \mathcal{D} = \{x_1 + x_2 \geq 1, \overline{x}_1 + \overline{x}_2 \geq 1\}$ we can use strengthening to infer $x_1 \geq 1$ with the witness $\omega = \{x_1 \mapsto x_2, x_2 \mapsto x_1\}$ swapping x_1 and x_2. This is so since $(\mathcal{C} \cup \mathcal{D})\restriction_\omega = \mathcal{C} \cup \mathcal{D}$ and $(x_1 \geq 1)\restriction_\omega = x_2 \geq 1$ follows by adding $\neg(x_1 \geq 1) = \overline{x}_1 \geq 1$ and $x_1 + x_2 \geq 1$ (and all these proof goals would be automatically inferred by the checker). Note, however, that if we continue by attempting to also derive $x_2 \geq 1$ in an analogous fashion, then this fails (as it should). Keeping track of whether all necessary proof obligations can be derived for strengthening rules is a crucial (and complicated) part of proof trimming.

Checked deletion. Deleting a constraint C from the derived set $\mathcal{D}$ is always allowed, but deleting from the core set $\mathcal{C}$ requires showing either $\mathcal{C} \setminus \{C\} \models C$ in strengthening-to-core mode or that C can be rederived by redundance-based strengthening from $\mathcal{C} \setminus \{C\}$ in regular mode.

Unchecked deletion. In regular mode, unchecked deletion of any core constraint is allowed.

Checked deletion ensures that removing constraints can never introduce spurious solutions, but any such guarantees are lost with unchecked deletion since the whole input formula $\mathcal{F}$ can just be erased. The reason for having stronger restrictions on deletion in strengthening-to-core mode is that arbitrary deletions from the core set would break the invariant $\mathcal{C} \models \mathcal{D}$.

III. TRIMMING VERIPB PROOFS

Our method to trim VERIPB proofs follows the three phases described below.

***Phase 1 (Decoration)**:* The first phase is a forward pass where we read the proof into memory and collect all information required to process the proof backwards in the next phase, including the constraint IDs for all derived constraints. This phase guarantees that for each line in the proof, we can reconstruct the proof configuration prior to this line from the proof configuration after the line. Because no propagation checks are performed, this phase is relatively cheap, but it can run into memory issues for very large proofs. Section V presents optimisations such as lazy evaluation of cutting planes derivations that can alleviate problems with memory.

***Phase 2 (Trimming)**:* In this phase, a backward pass over the proof *marks* all constraints that are necessary to refute the input formula (for simplicity, we focus here on unsatisfiability proofs—see Section V-B for a discussion of optimisation proofs). The process starts by marking the final contradiction in the proof. For every marked constraint, we identify its *antecedents*, i.e., constraints sufficient to derive it, which then become marked themselves. The process continues backwards through the proof, skipping any unmarked constraints. At the end of this process, the marked constraints constitute a valid smaller proof. For each rule in the VERIPB proof system, we identify the antecedents as follows:

`pol`. The cutting planes derivation explicitly lists the antecedents, which become marked.

`rup`. If a `rup` line is provided with antecedents (also called *annotations* or *hints*), those constraints become marked. Otherwise, the antecedents are extracted through pseudo-Boolean unit propagation and conflict analysis. Note that PB propagation requires access to all the constraints in the current configuration; this data was collected in the decoration phase. In Section V-A2 we discuss optimisations of the RUP algorithm for trimming, which can lead to smaller proofs and faster checking.

Deletions. *Unchecked* deletion of a constraint C is relocated to immediately after the last use of C, i.e., right after the line that marked C. In contrast, *checked* deletion of C cannot obviously be moved, since the rederivation of C by strengthening depends on the exact current proof

configuration, and so we leave such deletion steps in place.

Redundance-based strengthening. We discuss how to deal with strengthening in Section IV.

Phase 3 (Compaction): Finally the proof is cleaned up, leaving only the marked constraints; the deletion rule applications are shifted to their final locations; and constraint IDs are updated (since they should be consecutive integers).

We remark that what is described above is not the only possible approach. Nukala et al. [71] developed a trimmer for VERIPB proofs that only supports the implicational subset of the proof system. Their trimmer does not output a new VERIPB proof, but a proof in an intermediate format designed for translation into a clausal proof with a potential exponential blow-up in size.

IV. TRIMMING STRENGTHENING RULES

Strengthening rules, such as redundance-based strengthening (Section II) and dominance-based strengthening [17], introduce global dependencies that make trimming particularly convoluted.

When a strengthening rule is used, each constraint in the current configuration induces a proof obligation, and its corresponding proof, a *subproof*, may use other available constraints as antecedents. During the trimming phase, any marked proof obligations will cause their corresponding antecedents in the subproofs to be marked as well, and those newly marked antecedent constraints in turn induce their own proof obligations.

To illustrate this with a toy example, assume that, during a backward pass, the current set of constraints in the core set is $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ and that the derived set $\mathcal{D} = \emptyset$ is empty for simplicity. Assume also that the constraint C_m has been marked as needed in the trimmed proof but that no other constraint C_i for $i < m$ is currently marked. Suppose that in the current step C_m is inferred by redundance-based strengthening with witness ω. This means that we have to check that the proof obligations for strengthening can be met, but we only need to do so for currently marked constraints. When checking how to derive the proof obligation $C_m\restriction_\omega$, it could happen that the subproof uses C_{m-1} as a premise, which means that C_{m-1} needs to be marked. But then we also need to check the subproof for the proof obligation $C_{m-1}\restriction_\omega$. The subproof for this proof goal might happen to require C_{m-2}, so that this constraint should also be marked and we need to inspect the subproof for the proof obligation $C_{m-2}\restriction_\omega$. In this way, backward trimming of subproofs for proof obligations in strengthening steps can potentially cause a cascade of recursive markings until a fixpoint is reached. After reaching fixpoint, no further proof obligations need to be checked, because constraints that remain unmarked after reaching a fixpoint can either *(i)* not be included in the trimmed proof, and hence cause no proof obligations, or *(ii)* be safely deleted before the current proof step, since they are not needed later in the proof. This marking process is described in detail for clausal proofs in [75], where a more flexible strengthening rule is also introduced that renders this fixpoint computation unnecessary.

In the context of VERIPB proofs, however, such an approach does not work. In regular mode with strengthening-to-core switched off (as well as in DRAT and LRAT proofs), if a constraint is no longer needed then it can just be deleted, to ensure that it does not introduce proof obligations in later steps in the proof. This means that during the backward pass in regular mode we know exactly which constraints are proof obligations for each strengthening step. However, in strengthening-to-core mode, constraints can only be deleted from the core set using checked deletion,[2] which requires an explicit justification. Hence, absent such a justification for checked deletion of a core constraint C, the trimmer cannot just insert a deletion of C in the proof. In strengthening-to-core mode, therefore, it can only be known whether a core constraint is a proof obligation for a strengthening step once the backward pass has reached the location at which the proof mode was switched to strengthening-to-core, before which point unrestricted deletion is allowed.

This has severe implications for the marking process in strengthening rules: core constraints left unmarked by the fixpoint process described above may later be found to require marking. This then requires marking every antecedent in the subproof for the corresponding proof obligation, restarting the fixpoint process.

To tackle this situation, we introduce a finer-grained marking mechanism, which we call *conditional marking*. Instead of simply marking constraints, we collect *marking conditions* of the form $C_1, \ldots, C_n \Rightarrow C$; these state that if all constraints $C_1, \ldots, C_n$ are marked, then C must be marked as well. To trim a redundance rule deriving C through the witness ω while in strengthening-to-core mode, we check $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \models C'\restriction_\omega$ for *every* constraint $C' \in \mathcal{C} \cup \{C\}$, not just the marked ones. From this check, which is justified by an implicational proof, a set of antecedents can be extracted; each such antecedent C'' then contributes the marking condition $C, C' \Rightarrow C''$.

Example IV.1. Assume $\mathcal{C}$ contains the constraints $C_1 = a + 2x + 3y \geq 3$ and $C_2 = b + 2x + 3y \geq 3$. Consider a redundance rule that derives a constraint C using witness ω that swaps a and b (i.e., $\omega = \{a \mapsto b, b \mapsto a\}$). Then clearly $C_1\restriction_\omega = C_2$ and $C_2\restriction_\omega = C_1$. This produces the marking conditions $C, C_2 \Rightarrow C_1$ and $C, C_1 \Rightarrow C_2$, i.e., *if* C is present in the final trimmed proof, then either both of C_1 and C_2 should be included, or neither of them should.

We extend the trimming process to deal with marking conditions as follows.

- If a constraint C is *conditionally marked* through the marking condition $C_1, \ldots, C_n \Rightarrow C$, and D is an antecedent of C, we collect the marking condition $C \Rightarrow D$.
- Whenever all the left-hand side constraints to a marking condition $C_1, \ldots, C_n \Rightarrow C$ become unconditionally marked, we unconditionally mark C too. This process propagates unconditional markings until fixpoint.

[2]There are other situations where deletion is restricted and where the same problem arises; we come back to this in Section V-B1.

After the trimming phase, some conditionally marked constraints may remain unpropagated; these can all be safely considered unmarked. By propagating unconditional markings until fixpoint, any remaining conditionally marked constraints can only have as conditions constraints that are either unnecessary for the trimmed proof, or conditionally marked themselves. Omitting such constraints results in a valid proof; these are removed during the ensuing *compaction* phase.

Example IV.2 (Example IV.1 continued)**.** Assume that at the end of the trimming phase, C is unconditionally marked, but neither C_1 nor C_2 is marked. As argued above, we can then choose to either include both or neither of them in the final proof; since our goal is to trim the proof, the latter is chosen.

As explained above, conditional marking is only needed in contexts where deletion is restricted, like when strengthening-to-core mode is enabled. Elsewhere, the fixpoint approach can be used.

V. Optimisations and Extensions

In this section, we proceed to discuss trimming performance optimisations, how to trim applications of the dominance rule, and extensions to proofs for optimisation problems.

A. Optimisations of Proof Trimming Performance

The main optimisations included in our implementation of pseudo-Boolean proof trimming are to avoid storing multiple copies of the same constraint; to use prioritised propagation for `rup` rule applications (similarly to what is done in DRAT-TRIM [91]); and to improve the efficiency of checked deletion. Further optimisations are primarily motivated by the need to handle large proofs efficiently. We introduce what we call *lazy* `pol` *evaluation* and a *no-cache technique* to bound memory usage. We conclude the section by describing unchecked trimming and the *agnostic trimming* mode, which trade correctness checking for speed when a verified checker is used.

1) Multiple Copies of a Constraint: In a VeriPB proof, it is possible to derive the same constraint multiple times (with each new copy receiving its own identifier). This can be convenient due to the *interference* phenomenon discussed above. When applying strengthening rules, each constraint in the proof configuration imposes additional proof obligations. Therefore, it can be necessary to delete constraints in order for strengthening steps to be valid. If such deleted constraints are again needed later, they must be rederived.

However, in some solver-generated proofs, constraints are sometimes rederived for no obvious purpose. This is typically due to automated tooling that generates parts of the proof without checking for duplicates. Our trimmer optimises such proofs by ensuring that in any configuration encountered, at most one copy of each constraint is present in the core and the derived sets, respectively. If a proof derives two contemporaneous copies of the same constraint, only the first derivation is kept.

Algorithm 1 Prioritised Propagation

```
1: Input: D: constraint to be derived by RUP, F: current database, ρ: trail
2: M ← Marked(F) ∪ {¬D}
3: while True do
4:     if ∃C ∈ M : slack(C; ρ) < 0
5:         return CONFLICT                    ▷ Conflict with constraint in M
6:     if ∃C ∈ M, (a, ℓ) ∈ Terms(C) : a > slack(C; ρ) ∧ (ρ(ℓ) = ∗)
7:         ρ.push(ℓ, C)                       ▷ Propagate literal from constraint in M
8:         continue
9:     if ∃C ∈ F : slack(C; ρ) < 0
10:        return CONFLICT
11:    if ∃C ∈ F, (a, ℓ) ∈ Terms(C) : a > slack(C; ρ) ∧ (ρ(ℓ) = ∗)
12:        ρ.push(ℓ, C)
13:    else
14:        return FAILED
```

2) Prioritised Propagation: Ideally, trimmed VeriPB proofs should only contain a minimal sequence of steps necessary to reach the verdict stated in the final `CONCLUSION` section of the proof (though, as mentioned in the introduction, no theoretical guarantees can be provided in this regard for efficiency reasons). Our trimmer tries to minimise the marking of new constraints by using *prioritised propagation*. This method extends the *core-first propagation* technique used in clausal proofs [46] to the pseudo-Boolean setting. In this approach, marked constraints and constraints from the input formula are given precedence over unmarked ones in the propagation phase during a reverse unit propagation check. Consequently, new markings are only elicited when propagation using the currently marked constraints does not suffice to reach a conflict. Algorithm 1 gives pseudocode for this prioritised propagation. In this algorithm, lines 4–8 perform conflict detection and propagation over marked constraints. If this does not yield any new inferences, lines 9–12 search for a conflict or single propagation step over an unmarked constraint. Line 14 reports a failed RUP check if no further propagation is possible.

Unit propagation collects propagated literals in a stack called the *trail*. During the backward trimming pass, a large prefix of the trail can often be preserved from one RUP check to the next, and recomputing the trail from scratch can be computationally expensive. Keeping the trail is sound as long as all constraints propagating literals on the trail remain marked in the constraint database. However, the interaction between priority-sensitive propagation and trail reuse under dynamic database modifications (e.g., constraint deletions or changes in marking status) is nontrivial. In our current setting we cannot guarantee that an unmarked constraint will never propagate in a situation where a marked constraint could have propagated instead. Ensuring both trail reuse and strict adherence to priority order, similarly to the methods described in [76] for clausal proofs, requires a more thorough formal treatment, which we have to leave for future work.

3) Selective Checked Deletion: During VeriPB proof checking, the default behaviour of the checker is to delete constraints from the core set $\mathcal{C}$ using checked deletion until this fails, at which point the checker switches to unchecked deletion. Checked deletions can be computationally expensive,

and so are desirable to avoid whenever possible. Therefore, the trimmer only performs checked deletions when the proof state requires it, i.e., when a nontrivial order is loaded (see Section V-B further below) or strengthening-to-core is enabled. Otherwise, the trimmer deletes constraints from $\mathcal{C}$ using unchecked deletions. If a compulsory checked deletion fails, the proof is rejected as invalid.

4) Techniques for Bounding Memory Usage: Optimising the usage of memory during trimming is required because VERIPB proofs exhibit two opposing structural phenomena.

- **Compact proofs with massive constraint databases.** In contrast to SAT proofs, which list clauses explicitly, VERIPB proofs may encode constraints compactly by specifying the cutting planes derivations used to infer them. This means that although the size of the original proof can be moderate, it can become much larger when the explicit constraints are written out during the decoration phase. As a concrete example, we identified a 273 MB VERIPB proof instance generated by the GLASGOWSUBGRAPHSOLVER [65] that resulted in a trimmer memory usage of 50 GB due to the derived constraints. Thus, even if proof logging has been implemented to minimise disk usage, the memory footprint when generating explicit constraints during trimming can become a significant bottleneck.
- **Verbose proofs with small constraints.** Conversely, a cutting planes derivation in a single `pol` line may involve many primitive instructions, ultimately producing a very small constraint. For example, during preliminary experiments we observed a constraint on 8 variables that the SAT solver CADICAL [20] derived using 96,466 instructions. In such cases, memory pressure does not stem from the constraint database itself, but rather from storing and managing the parsed instruction sequences.

These two extremes require distinct optimisation strategies. We address the first using *lazy `pol` evaluation*, and the second using the *no-cache strategy*.

- **Lazy `pol` evaluation:** We store reverse Polish notation `pol` steps in symbolic form, evaluating them only when explicitly referenced by a marked step rather than when they are first found. This bounds memory usage by the size of the set of relevant constraints rather than the size of the entire accumulated database.
- **No-cache strategy:** To handle verbose proofs, parsed `pol` lines are discarded immediately after processing rather than cached. Such lines are re-parsed on demand, minimising the memory footprint of the proof structure itself.

In order to execute certain VERIPB proof rules, the trimmer needs to know not only about constraint IDs but also the exact syntactic form of each (currently undeleted) constraint. In particular, proof steps that force the trimmer to search over the whole constraint database make the trimmer revert temporarily to eager evaluation of `pol` steps (expanding all constraints not yet deleted), which diminishes the performance benefits of trimming. For the benefit of expert readers, we note that the following constructs exhibit this behaviour:

- **Implicit steps:** `rup` steps without annotations/hints (and that rely on not-yet-marked constraints derived through `pol` steps), claims in the `CONCLUSION` section without explicit constraint references, and missing subproofs prevent the (effective) use of lazy `pol` evaluation.
- **Deletion:** To make it easier to translate SAT proofs to the VERIPB proof format, there is a *specification-based deletion* rule `del spec` that deletes a constraint not by referring to the constraint ID but by giving the syntactic form of the constraint. This is in conflict with lazy `pol` evaluation, because trimmer knowledge about the constraint database is inherently incomplete. When the trimmer encounters such a line, this forces an immediate evaluation of all pending symbolic `pol` lines to reconstruct the full constraint database. Frequent use of this feature degrades performance to be similar to when eager evaluation is used.

5) Trimming Without Checking: In situations where the trimmed proof will be checked using the formally verified backend CAKEPB, validating the proof during trimming is unnecessary to establish proof correctness. The *unchecked trimming* mode allows the trimmer to skip semantic verification, focusing solely on dependency analysis. Correctness is left to the downstream verified checker CAKEPB, allowing for significantly faster processing. Pushing the idea further, the *agnostic trimming* mode bypasses the initial forward decoration pass entirely, performing unchecked trimming in a strict back-to-front manner. This mode imposes several restrictions on input VERIPB proofs:

- The proof is treated as a stream of instructions to be marked, relying entirely on explicit hints. Hence, input proofs must consist solely of proof steps with fully specified antecedents (e.g., `pol` and hinted `rup` steps).
- Complex derivation rules (such as strengthening rules or solution logging rules) are not admissible.
- The only supported type of proofs are proofs of unsatisfiability.
- Since the trimmer lacks knowledge of the global constraint count during backward trimming, there are limitations on features such as referencing constraints by relative ID offsets rather than by absolute constraint IDs.

B. Extensions to More Advanced Proof Rules and Types

We now turn to a discussion of how the trimmer handles the *dominance-based strengthening* rule, or just *dominance* for brevity, and proofs for optimisation problems. These are two of the most complex features of VERIPB.

More formally speaking, a VERIPB proof for a formula $\mathcal{F}$ with objective function f can be represented as a sequence of proof steps updating a proof configuration $(\mathcal{C}, \mathcal{D}, \mathcal{O}_{\preceq}, \vec{z}, f, v)$ consisting of

- $\mathcal{C}$ and $\mathcal{D}$, the sets of core and derived PB constraints;
- a PB formula $\mathcal{O}_{\preceq}$ encoding a (possibly trivial) preorder on assignments to an ordered set of literals $\vec{z}$;

- an objective function f together with the best value v found for it so far.

We will need to refer to this more formal setting in what follows.

1) Dominance: The dominance rule `dom` makes use of an order specified by a pseudo-Boolean formula $\mathcal{O}_{\preceq}(\vec{u}, \vec{v}, \vec{a})$ over placeholder variable sets $\vec{u} = \{u_1, \ldots, u_n\}$ and $\vec{v} = \{v_1, \ldots, v_n\}$ of equal size together with an optional set of auxiliary variables $\vec{a}$. If auxiliary variables are used, they come with a specification $\mathcal{S}_{\preceq}$, which is a second PB formula over $\vec{u}, \vec{v}$, and $\vec{a}$; otherwise, no specification is required. For further details, we refer to [5], [6], [17].

A PB constraint C can be derived from $\mathcal{C} \cup \mathcal{D}$ by dominance if there exists a substitution ω such that both conditions

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \models \mathcal{C}\restriction_\omega \cup \mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \cup \{f\restriction_\omega \leq f\} \quad (1)$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \models \bot \quad (2)$$

can be shown to hold. Here $\vec{z}$ denotes the set of literals to which the order is applied (which is part of the configuration above). Intuitively, the witness gives us a mechanism to patch up any assignment that satisfies the current constraint database but violates the constraint that we wish to infer. However, in contrast to the redundance-based strengthening `red` rule, this patching may require repeated applications of the witness to reach a satisfying assignment. The order serves to guarantee that this process will eventually terminate, and that the final assignment will satisfy the derived constraint. For the full details of this rule including formal proofs of correctness, the reader is referred to [17].

To trim dominance rule applications, the subproof for each proof goal on the right-hand sides of (1) and (2) is processed using the conditional marking technique introduced in Section IV. The list of conditions for each constraint C'' used in such subproofs is updated as follows:

1) If C'' is used in the subproof for a proof obligation $C'\restriction_\omega$ with $C' \in \mathcal{C}$ such that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \models C'\restriction_\omega$$

holds, then we add the condition $C, C' \Rightarrow C''$. This case is analogous to the method used for trimming redundance-based strengthening rules in Section IV.

2) If C'' is used in establishing any of the proof obligations

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \models \mathcal{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a})$$
$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}, \vec{a}) \models \{f\restriction_\omega \leq f\}$$
$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{S}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \cup \mathcal{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega, \vec{a}) \models \bot$$

then we add the conditional marking $C \Rightarrow C''$. Unlike in the former case, here the constraints in $\mathcal{C}$ do not appear as proof goals, so only C is needed as a condition.

Dominance rules also give rise to additional proofs, on top of the rule applications, in that any order used in a `dom` rule must be *defined* through a dedicated proof section that specifies the pseudo-Boolean formula encoding the order and also includes subproofs showing that this formula yields a transitive and reflexive relation; we refer the interested reader to [6], [17]. Transitivity and reflexivity proofs use their own, separate constraint database, so trimming these proofs of order properties can be done independently of the backward trimming pass. Our trimmer handles transitivity and reflexivity proofs in order definitions during the compaction pass.

As noted in Section IV, the role of the compaction pass extends beyond merely writing the final proof to file. Specifically, this pass performs subproof checking and elaboration (i.e., determining missing antecedents); it also trims residual conditional markings and verifies and elaborates the order definition. While reflexivity can be automatically proven by the VERIPB checker if the proof does not specify a subproof for this property, automatic proving of transitivity is not supported.

2) Optimisation and Objective Functions: VERIPB proofs can also establish bounds on the optimal value of an objective function f subject to the constraints in an input formula $\mathcal{F}$. In such proofs, one can log a solution to the formula $\mathcal{F}$ in the proof to obtain a so-called *solution-improving constraint* that enforces that the next solution found must yield a better value than the incumbent. Additionally, to support solver preprocessing techniques there are rules for modifying the objective function, which require a proof that the new objective evaluates to the same value as the old one for any solution satisfying the constraint database. The trimmer must ensure that the trimmed proof establishes the same bounds as the original proof, which requires special handling of these rules.

a) Solution-Improving Constraints: The *solution-improving* `soli` rule requires a specification of a solution α satisfying the constraints in the database. After checking this solution, a new *solution-improving constraint* $f \leq f\restriction_\alpha - 1$ is added to the core set $\mathcal{C}$. For proof trimming, the exact syntactic form of this constraint must be inferred during the decoration pass, so that it can be included in $\mathcal{C}$ at the start of the backward trimming.

It is possible to specify the solution by a partial assignment ρ in a `soli` step, provided that the full solution α can be reconstructed from ρ by unit propagation over the current constraint database. To be able to determine the exact form of the solution-improving constraints during the decoration phase (in which no unit propagation is performed), we require that proofs to be trimmed should satisfy one of the following two conditions:[3]

1) Each `soli` line is followed by a so-called *equals* or `e` line that specifies the explicit form of the solution-improving constraint just added. The `e` rule has no effect on the proof state, but is a convenience rule used to check that a specific constraint ID refers to a specific syntactic constraint as specified in the rule application. This can be useful for communicating information to the trimmer, which might not have access to the full proof state.

[3]In all proof logging applications we are aware of, these conditions are either already satisfied by the produced proofs, or are easy to satisfy by slightly modifying the proof logging.

2) The assignment in each `soli` line is a complete assignment over the objective variables.

If the `soli` line is not followed by an `e` line, the trimmer assumes that condition 2 applies. If this assumption is violated, the proof is rejected.

For each checked `soli` line with an ensuing `e` line, the backward trimmer reconstructs the full assignment α obtained by propagation starting from ρ, so that the `soli` lines in the final trimmed proof will specify a full assignment as required by the formally verified backend checker CAKEPB.

b) Conclusion Proof Sections with Objective Bounds: In VERIPB proofs of unsatisfiability, the final `CONCLUSION` section of the proof specifies the constraint ID of a contradictory constraint (such as $0 \geq 1$), which is used as the first marked constraint in the backward trimming phase. Proofs for optimisation problems instead state upper and lower bounds on the objective function to be minimised and optionally a solution witnessing the upper bound. For such proofs, the first marked constraint will be the constraint specified as proof of the lower bound on the objective. The upper bound is handled as follows:

1) If the `CONCLUSION` section provides a solution to $\mathcal{F}$, i.e., a (partial) assignment that satisfies all constraints in $\mathcal{F}$, the backward trimmer accepts the upper bound and the compaction phase logs the full solution in the `CONCLUSION` section of the trimmed proof.
2) Otherwise, a solution witnessing the upper bound must be provided in the proof proper. This means that the proof can only erase constraints by checked deletion up to the point when a solution witnessing the upper bound is logged with a `soli` step. This is sufficient since checked deletion maintains *equioptimality*, i.e., that the objective function has the same optimal value over the current constraint database as over the input formula. As a consequence of this, the backward trimmer will force checked deletion to be used for all deletions that precede the logging of the best solution. During the compaction phase, it will also print the full solution (which is reconstructed from the `soli` step) in the `CONCLUSION` section of the trimmed proof.

Since the trimmer will guarantee that a full solution to the original formula is specified in the `CONCLUSION` section at the end of the proof, the trimmed proof is not required to maintain equioptimality. This makes it possible to use unchecked deletions, which improves proof checking speed.

c) Objective Function Updates: VERIPB offers two rules `obju diff` and `obju new` to update the objective function from the current objective f_{curr} to a new objective f_{new}. For both of these rules, it is required to demonstrate that this change of objective does not change the optimal value of the problem by deriving $f_{curr} = f_{new}$ (in the form of the two inequalities $f_{new} \geq f_{curr}$ and $f_{curr} \geq f_{new}$) from the core set of constraints $\mathcal{C}$. The difference between the two rules is in how the new objective is specified: the `obju new` rule lists the new objective function f_{new} explicitly, whereas the `obju diff` rule states the difference $f_{new} - f_{curr}$, from which the checker can infer f_{new} since it knows f_{curr}. The latter rule is better if a large objective function is rewritten in many small steps by adding or removing terms, since in such a scenario it decreases proof size and increases checking speed.

In principle, we could trim objective update rule applications that are not needed in the final proof. However, if the final upper bound is logged with a `soli` step adding the constraint $f_{new} \leq f_{new}\restriction_{\alpha} - 1$, and if optimality is then proven by using this solution-improving constraint, then the objective update cannot be trimmed away. In practice, we do not try to trim away objective update rule applications, because every change from f_{curr} to f_{new} creates a dependency between the old and new objective, and this quickly leads to a chain of dependencies. If the final `soli` step makes use of the last objective update, then this means we have to preserve the entire sequence of objective function modifications.

VI. EXPERIMENTAL RESULTS

All the techniques described in this paper have been implemented in Rust as a new mode of the existing VERIPB proof checker [5], [90], submitted as an accompanying artefact [1]. The aim of our tool is to provide a fully featured trimmer that can handle the full extent of rules and proof structures supported by the VERIPB 3.0 proof format. To demonstrate that we achieved this, and to evaluate the reductions in proof size and proof checking time achieved by our trimmer, we present results from computational experiments on a large set of representative proofs. The proof files were obtained by running six different combinatorial solvers from across the spectrum of problem paradigms, all of which have been designed or retrofitted with VERIPB proof logging. Wherever possible, the input problem instances were taken from benchmark sets previously used in publications evaluating proof logging performance for the respective solvers. Parameters for proof production were chosen based on available disk space and rate of proof writing for each solver. Specifically:

- The GLASGOWSUBGRAPHSOLVER (GSS) [65] with proof logging added in [39], [41] was run on the LV instances [58] from a standard set of subgraph isomorphism benchmarks [81], [82]. We restricted the number of vertices to 260 in the target graph and set a time limit of 100 s for GSS without proof generation.
- The GLASGOWCLIQUESOLVER (GSS-CLIQUE) is a separate solver included in the GSS repository, and we ran this with clique instances from [39].
- The GLASGOWCONSTRAINTSOLVER (GCS) [42] was run on the randomly generated instances from [67], [69].
- The PACOSE MaxSAT solver [72] with proof logging added in [10] was run on instances from the MaxSAT Evaluation 2023 [61] with a time limit of 1000 s.
- The ROUNDINGSAT pseudo-Boolean solver [27], [28], [32], for which optimised proof logging was implemented in [56], was run on instances from the Pseudo-Boolean Competition 2024 [73] with a time limit of 300 s.

- A version KISSAT-SATSUMA of the KISSAT SAT solver [54] extended with SATSUMA symmetry breaking [7] and required proof logging changes [6] was run on unsatisfiable instances from the SAT Competition 2024 [79] with a time limit of 1000 s.

This resulted in a total of 2656 instance-proof file pairs, using 2.7 TB of disk space. The benchmarks were highly diverse in terms of generated proof size, from proofs consisting of only a few lines to proofs larger than 80 GB. We are also confident that the proofs together make use of almost all available VERIPB features, including strengthening-to-core mode, dominance rule applications with orders, objective function updates, proof steps with and without hints/annotations, and different degrees of reliance on implicit derivations by `rup` or explicit cutting planes derivations with `pol`.

For each instance we tested both our trimmer and the forward-checking VERIPB elaborator, and then ran the CAKEPB verified checker on the trimmed and elaborated proofs, respectively. This is representative of a typical formally verified proof checking workflow. Each run was performed on a cluster of machines with dual AMD EPYC 7643 CPUs, 2 TB of RAM, local solid state drives, and Ubuntu 25.10 as operating system. We limited the available RAM for each process to 40 GiB.

We present in Figure 1 the result of plotting the total time of elaboration + CAKEPB checking against trimming + CAKEPB checking. A plot of proof size comparisons is then shown in Figure 2 and average ratios for each solver are given in Table I. To avoid giving an unfairly positive view of the achievements of our trimmer, there are several instances that were excluded from the table. The excluded instances are:

- 54 instances where the trimming pipeline finished, but the elaboration pipeline timed out,
- 182 proofs with a conclusion `SAT` or `NONE` (where proof trimming would remove the whole proof),
- 166 proofs with open bounds from incomplete solver runs (where much of the proofs could be discarded).

The latter two categories are also excluded from the plots.

Our results show that trimming instead of just elaborating proofs can consistently improve checking time by an average factor of 1.81 and reduce proof size by an average factor of 6.37. Some solvers benefit more from trimming than others. For example, proofs for clique problems are nearly unchanged in size after trimming. In contrast, ROUNDINGSAT and GLASGOWCONSTRAINTSOLVER proofs shrink by factors of 11.1 and 17.1, respectively, suggesting that pseudo-Boolean and constraint programming solvers tend to produce more verbose proofs with lots of room for trimming.

A few proofs grow bigger after trimming—the number of lines is smaller, but proof size is larger due to `rup` hints. To illustrate how this can happen, assume that we have the constraints $\overline{x}_i + x_{i+1} \geq 1$ and $\overline{x}_N + y_i \geq 1$ for $i = 1, \ldots, N$ in the database. Then the proof can infer $\overline{x}_1 + x_N \geq 1$ and then derive N constraints $\overline{x}_1 + y_i \geq 1$ by RUP (which just requires propagating on $\overline{x}_1 + x_N \geq 1$ and $\overline{x}_N + y_i \geq 1$).

Solver	time[1]	time[2]	time[3]	size[4]
GSS	1.62	3.36	2.29	6.85
ROUNDINGSAT	1.39	2.70	2.02	11.12
PACOSE	0.95	2.73	1.59	4.22
KISSAT-SATSUMA	1.32	1.70	1.45	1.74
GCS	1.54	2.07	1.68	17.13
GSS-CLIQUE	1.00	1.01	1.01	1.11
all	1.27	2.57	1.81	6.37

TABLE I: 1-shifted geometric means: [1] Initial elaboration / trimming time ratio. [2] CAKEPB verification time ratio (elaborated / trimmed). [3] Total check time ratio (elaborated / trimmed). [4] Proof size ratio (elaborated / trimmed).

Suppose now that these `rup` lines are without hints, and that the constraint $\overline{x}_1 + x_N \geq 1$ is trimmed away. Then all RUP checks still pass, but the N `rup` lines in the elaborated proof will have N hints each (instead of 2 hints each, which would be possible when elaborating without trimming). Such phenomena were observed rarely in our experiments, which indicates that reducing the number of proof lines is a good heuristic for trimming in general, but it also raises the question of what other approaches for trimming could be useful.

VII. CONCLUDING REMARKS

Although proof logging has long been a success story for SAT solving, with proof formats and tools that allow proof logging and checking with small overhead, progress on certifying solvers for combinatorial solving paradigms beyond SAT has been much more limited. This has started to change in the last few years with the advent of pseudo-Boolean proof logging using VERIPB, but the efficiency of generating and validating such more expressive proofs has remained a serious concern. In particular, no general method has been available for trimming pseudo-Boolean proofs, whereas proof trimming is crucial for efficient formally verified checking in SAT.

In this work, we present for the first time a proof trimming tool for fully general pseudo-Boolean proofs in the VERIPB proof logging system. Designing and implementing such a tool involves a complex interplay between identifying as many constraints as possible for removal and ensuring that applications of the sophisticated strengthening and deletion rules remain sound. In contrast to clausal SAT proofs, pseudo-Boolean proof trimming requires several passes over the proof, as well as more involved conditional marking of constraints for which it can only be decided later whether they are necessary to keep in the proof or not.

We evaluate our trimmer on proofs generated in a diverse range of combinatorial solving paradigms, each involving different technical challenges, and demonstrate that our general trimming method succeeds in significantly reducing VERIPB proof size across the board. Even more importantly, trimming also decreases the time required to *(1)* elaborate proofs and then *(2)* validate them with the formally verified proof checker CAKEPB.

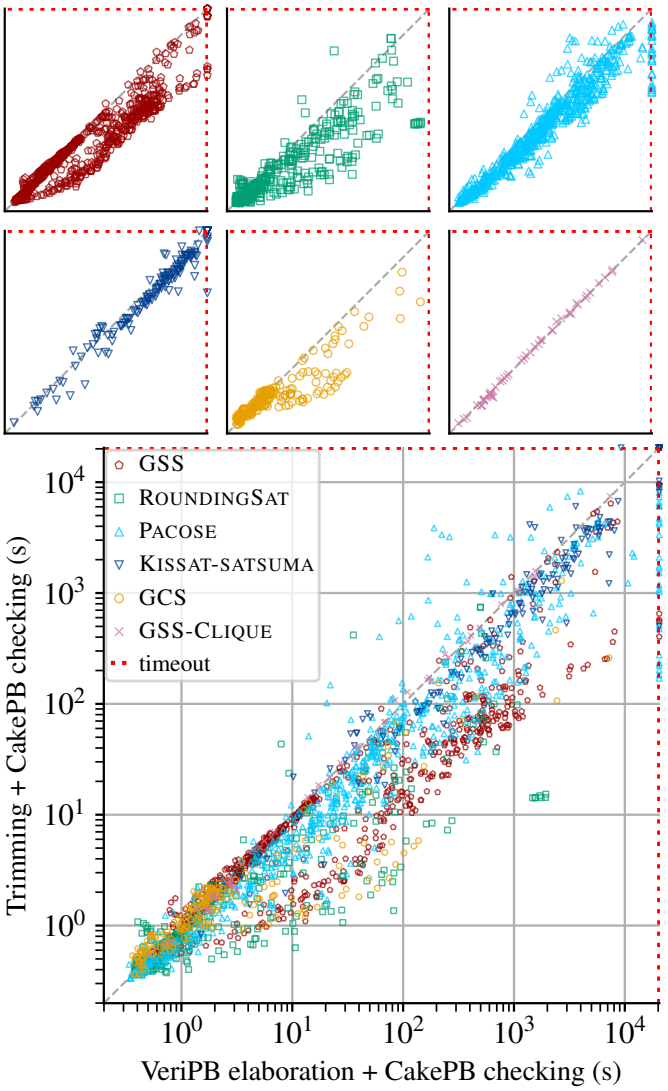

Fig. 1: Time comparison for trimming and elaboration pipelines. The timeout lines represent those instances where any stage of the respective pipeline of tools timed out.

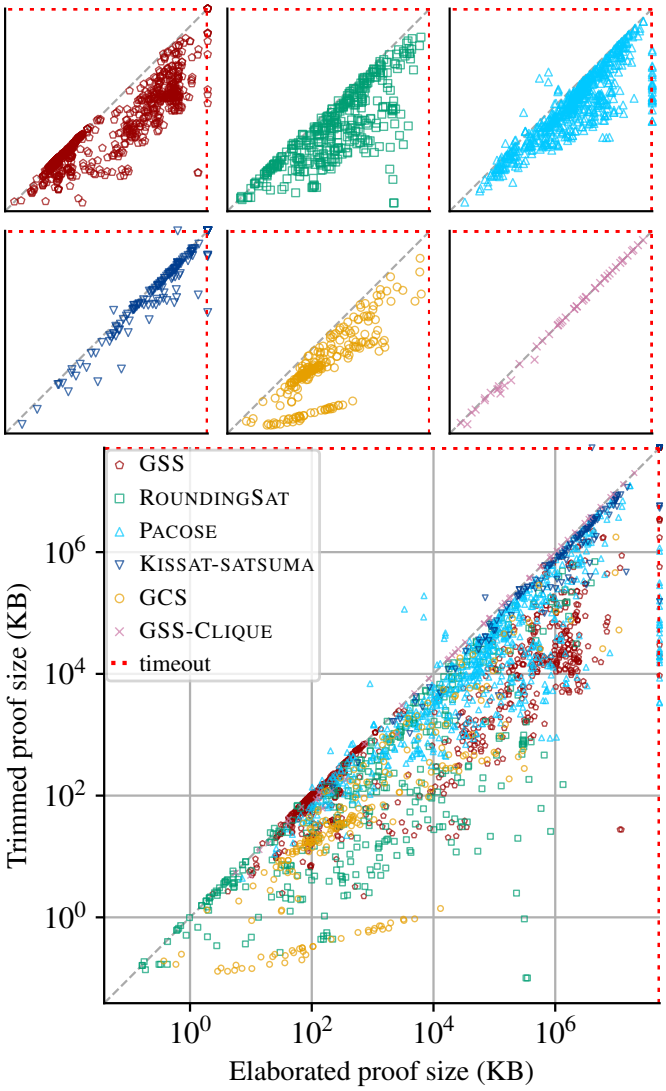

Fig. 2: Proof size comparison for trimming and elaboration.

We see many interesting directions for future work. In particular, since the task of marking constraints to be kept in the proof is quite challenging, there is room for developing better heuristics for proof steps where the checker needs to perform propagations on the whole constraint database. One idea is to prioritise constraints derived earlier in the proof, since they have fewer potential antecedents. Another choice could be to prefer constraints with explicit derivations (e.g., via `pol` or hinted `rup` lines), where the information about which premise constraints to mark is cheap to extract. Yet another option could be to prioritise strength of constraints during the marking process (e.g., preferring strong cardinality constraints to long, weak, disjunctive clauses). We would like to understand better the interaction between different heuristics in terms of proof size, trimming time, and checking time. This can also shed more light on which classes of proofs benefit most from trimming, and which optimisations are most effective.

Our current trimmer is optimised for trimming speed, but for some applications (such as proof storage) one may want to spend more time to extract even smaller proofs. This could also be used to identify more precisely why a problem is unsatisfiable (i.e., to extract small unsatisfiable cores).

We believe our trimmer could also be adapted to support work on *proof skeletons*, which have been proposed for decreasing proof logging overhead [34], [77]. The idea is to initially produce a proof sketch (with gaps). Trimming can then identify which parts of this sketch are needed, after which a second pass can fill in the missing intermediate steps.

A final, important, direction for future work is trimming of very large proofs. Our current trimmer reads and stores the whole proof in memory. To trim large proof files in a streaming fashion, the proof format would need to support even more precise annotations to make it possible to recover all required proof details during a backward pass through the proof.

References

[1] Berhan Oumer Adame, Bart Bogaerts, Benjamin Bogø, Simon Dold, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, Adrian Rebola-Pardo, and Mark Turnbull. Artefact for: "Trimming pseudo-Boolean proofs". https://doi.org/10.5281/zenodo.20136181, May 2026.

[2] Özgür Akgün, Jessica A. Enright, Christopher Jefferson, Ciaran McCreesh, Patrick Prosser, and Steffen Zschaler. Finding subgraphs with side constraints. In *Proceedings of the 18th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '21)*, volume 12735 of *Lecture Notes in Computer Science*, pages 348–364. Springer, July 2021.

[3] Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, August 2018.

[4] Cyril Allignol, Nicolas Barnier, Pierre Flener, and Justin Pearson. Constraint programming for air traffic management: A survey. *The Knowledge Engineering Review*, 27(3):361–392, July 2012.

[5] Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Documentation of VeriPB and CakePB. Available at https://www.cril.univ-artois.fr/PB26/descr/Documentation_VeriPB_general.pdf, June 2026.

[6] Markus Anders, Bart Bogaerts, Benjamin Bogø, Arthur Gontier, Wietze Koops, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Adrián Rebola-Pardo, and Yong Kiam Tan. Faster certified symmetry breaking using orders with auxiliary variables. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, pages 14140–14148, January 2026.

[7] Markus Anders, Sofia Brenner, and Gaurav Rattan. Satsuma: Structure-based symmetry breaking in SAT. *Journal of Artificial Intelligence Research*, 85:6:1–6:31, January 2026. Preliminary version in *SAT '24*.

[8] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD '18)*, pages 1–9, October-November 2018.

[9] Anton Belov, Marijn J. H. Heule, and João P. Marques-Silva. MUS extraction using clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 48–57, July 2014.

[10] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Certifying without loss of generality reasoning in solution-improving maximum satisfiability. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, September 2024.

[11] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 1–22. Springer, July 2023.

[12] Sidhant Bhavnani. SmolPB: Trimming pseudo-Boolean proof logs. MSci thesis, University of Glasgow, April 2023.

[13] Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 10, pages 277–303. Springer, December 2018.

[14] Nikolaj Bjørner and Leonardo de Moura. The inner magic behind the Z3 theorem prover, October 2019. Blogpost at https://www.microsoft.com/en-us/research/blog/the-inner-magic-behind-the-z3-theorem-prover/.

[15] Ignace Bleukx, Ryma Boumazouza, Tias Guns, Nadine Laage, and Guillaume Poveda. Modeling and explaining an industrial workforce allocation and scheduling problem. In *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP '25)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:24, August 2025.

[16] Ignace Bleukx, Maarten Flippo, Emir Demirović, Bart Bogaerts, and Tias Guns. Using certifying constraint solvers for generating step-wise explanations. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, pages 14192–14200, January 2026.

[17] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, August 2023. Preliminary version in *AAAI '22*.

[18] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, pages 233–350. IOS Press, 2nd edition, February 2021.

[19] Samuel R. Buss and Neil Thapen. DRAT proofs, propagation redundancy, and extended resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 71–89. Springer, July 2019.

[20] CaDiCaL simplified satisfiability solver. https://github.com/arminbiere/cadical.

[21] Kevin K. H. Cheung, Ambros M. Gleixner, and Daniel E. Steffy. Verifying integer programming results. In *Proceedings of the 19th International Conference on Integer Programming and Combinatorial Optimization (IPCO '17)*, volume 10328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, June 2017.

[22] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[23] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, August 2017.

[24] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. 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)*, volume 10205 of *Lecture Notes in Computer Science*, pages 118–135. Springer, April 2017.

[25] Emir Demirović, Ciaran McCreesh, Matthew McIlree, Jakob Nordström, Andy Oertel, and Konstantin Sidorov. Pseudo-Boolean reasoning about states and transitions to certify dynamic programming and decision diagram algorithms. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, September 2024.

[26] Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. A scalable algorithm for minimal unsatisfiable core extraction. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT '06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 36–41. Springer, August 2006.

[27] Jo Devriendt, Ambros Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-Boolean conflict-driven search. *Constraints*, 26(1–4):26–55, October 2021. Preliminary version in *CPAIOR '20*.

[28] Jo Devriendt, Stephan Gocht, Emir Demirović, Jakob Nordström, and Peter Stuckey. 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)*, pages 3750–3758, February 2021.

[29] Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger, and Tanja Schindler. Pseudo-Boolean proof logging for optimal classical planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS '25)*, pages 54–63, November 2025.

[30] Simon Dold, George Katsirelos, Wietze Koops, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end certified graph colouring. In *Proceedings of the 32nd International Conference on Principles and Practice of Constraint Programming (CP '26)*, pages 21:1–21:27, July 2026.

[31] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, pages 1486–1494, February 2020.

[32] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In *Proceedings of the 27th International Joint*

Conference on Artificial Intelligence (IJCAI '18), pages 1291–1299, July 2018.

[33] Andreas Falkner, Gerhard Friedrich, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner. Twenty-five years of successful application of constraint technologies at Siemens. *AI Magazine*, 37(4):67–80, 2016.

[34] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A multi-stage proof logging framework to certify the correctness of CP solvers. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, September 2024.

[35] Graeme Gange, Geoffrey Chu, and Peter J. Stuckey. Certifying optimality in constraint programming. Manuscript. Available at https://people.eng.unimelb.edu.au/pstuckey/papers/certified-cp.pdf, 2023.

[36] Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, October 2019.

[37] Stephan Gocht. *Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning*. PhD thesis, Lund University, June 2022. Available at https://portal.research.lu.se/en/publications/certifying-correctness-for-combinatorial-algorithms-by-using-pseu.

[38] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Certified CNF translations for pseudo-Boolean solving. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:25, August 2022.

[39] Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. 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)*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, September 2020.

[40] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[41] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, pages 1134–1140, July 2020.

[42] Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. An auditable constraint programming solver. In *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP '22)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, August 2022.

[43] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI '21)*, pages 3768–3777, February 2021.

[44] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, pages 886–891, March 2003.

[45] Marijn Heule and Benjamin Kiesl. The potential of interference-based proof systems. In *1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (ARCADE '17)*, volume 51 of *EPiC Series in Computing*, pages 51–54, August 2017. Available at https://easychair.org/publications/paper/P113.

[46] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, pages 181–188, October 2013.

[47] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, June 2013.

[48] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, July 2016.

[49] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Certified MaxSAT preprocessing. In *Proceedings of the 12th International Joint Conference on Automated Reasoning (IJCAR '24)*, volume 14739 of *Lecture Notes in Computer Science*, pages 396–418. Springer, July 2024.

[50] Hannes Ihalainen, Dieter Vandesande, André Schidler, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Efficient and reliable hitting-set computations for the implicit hitting set approach. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, pages 14251–14260, January 2026.

[51] Christoph Jabs, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Certifying Pareto optimality in multi-objective maximum satisfiability. In *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '25)*, volume 15697 of *Lecture Notes in Computer Science*, pages 108–129. Springer, May 2025.

[52] Christoph Jabs, Jeremias Berg, and Matti Järvisalo. Engineering and evaluating multi-objective pseudo-Boolean optimizers. In *Proceedings of the 19th European Conference on Logics in Artificial Intelligence (JELIA '25)*, volume 16093 of *Lecture Notes in Computer Science*, pages 115–134. Springer, September 2025.

[53] Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. A SAT attack on Rota's basis conjecture. In *Proceedings of the 25th International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, August 2022.

[54] The Kissat SAT solver. https://github.com/arminbiere/kissat.

[55] Janne I. Kokkala and Jakob Nordström. Using resolution proofs to analyse CDCL solvers. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP '20)*, volume 12333 of *Lecture Notes in Computer Science*, pages 427–444. Springer, September 2020.

[56] Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, and Marc Vinyals. Practically feasible proof logging for pseudo-Boolean optimization. In *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP '25)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:27, August 2025.

[57] Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, March 2020. Extended version of paper in *CADE* 2017.

[58] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4):403–422, August 2002.

[59] Duc Anh Le, Stéphanie Roussel, and Christophe Lecoutre. Aircraft resource-constrained assembly line balancing with learning effect: A constraint programming approach. In *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming (CP '25)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:24, August 2025.

[60] Pey-Chang Kent Lin and Sunil P. Khatri. Application of Max-SAT-based ATPG to optimal cancer therapy design. *BMC Genomics*, 13(Suppl 6), October 2012.

[61] MaxSAT evaluation 2023. https://maxsat-evaluations.github.io/2023, July 2023.

[62] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[63] Ciaran McCreesh, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Proof logging for projected enumeration (and counting?) problems in VeriPB. In *Proceedings of the 32nd International Conference on Principles and Practice of Constraint Programming (CP '26)*, pages 43:1–43:21, July 2026.

[64] Ciaran McCreesh, William Pettersson, and Patrick Prosser. Understanding the empirical hardness of random optimisation problems. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *Lecture Notes in Computer Science*, pages 333–349. Springer, September 2019.

[65] Ciaran McCreesh, Patrick Prosser, and James Trimble. The Glasgow subgraph solver: Using constraint programming to tackle hard subgraph isomorphism problem variants. In *Proceedings of the 13th International Conference on Graph Transformation (ICGT '20)*, volume 12150 of *Lecture Notes in Computer Science*, pages 316–324. Springer, June 2020.

[66] Matthew McIlree. *Pseudo-Boolean Proof Logging for Constraint Propagation Algorithms*. PhD thesis, University of Glasgow, June 2026. Available at https://theses.gla.ac.uk/86049/.

[67] Matthew McIlree and Ciaran McCreesh. Proof logging for smart extensional constraints. In *Proceedings of the 29th International Conference on Principles and Practice of Constraint Programming (CP '23)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, August 2023.

[68] Matthew McIlree and Ciaran McCreesh. Certifying bounds propagation for integer multiplication constraints. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI '25)*, pages 11309–11317, February-March 2025.

[69] Matthew McIlree, Ciaran McCreesh, and Jakob Nordström. Proof logging for the circuit constraint. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14743 of *Lecture Notes in Computer Science*, pages 38–55. Springer, May 2024.

[70] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Emma Rollon. Employee scheduling with SAT-based pseudo-Boolean constraint solving. *IEEE Access*, 9:142095–142104, October 2021.

[71] Karthik V. Nukala, Soumyaditya Choudhuri, Randal E. Bryant, and Marijn J. H. Heule. Translating pseudo-Boolean proofs into Boolean clausal proofs. In *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design (FMCAD '24)*, pages 175–185, October 2024.

[72] Tobias Paxian, Sven Reimer, and Bernd Becker. Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, pages 37–53. Springer, July 2018.

[73] Pseudo-Boolean competition 2024. https://www.cril.univ-artois.fr/PB24/, August 2024.

[74] Xavier Pucel and Stéphanie Roussel. Constraint programming model for assembly line balancing and scheduling with walking workers and parallel stations. In *Proceedings of the 30th International Conference on Principles and Practice of Constraint Programming (CP '24)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:21, September 2024.

[75] Adrián Rebola-Pardo. Even shorter proofs without new variables. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT '23)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:20, July 2023.

[76] Adrián Rebola-Pardo and Luís Cruz-Filipe. Complete and efficient DRAT proof checking. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design (FMCAD '18)*, pages 1–9, October-November 2018.

[77] Joseph E. Reeves, Haniel Barbosa, Andrew Reynolds, and Marijn J. H. Heule. A general approach for SMT proof skeletons. In *Proceedings of the 13th International Joint Conference on Automated Reasoning (IJCAR '26)*, volume 16688 of *Lecture Notes in Computer Science*, pages 156–174. Springer, July 2026.

[78] Neha Rungta. A billion SMT queries a day (Invited paper). In *Proceedings of the 34th International Conference on Computer Aided Verification (CAV '22)*, volume 13372 of *Lecture Notes in Computer Science*, pages 3–18. Springer, August 2022.

[79] SAT Competition 2024. https://satcompetition.github.io/2024/.

[80] Laurent Simon. Post mortem analysis of SAT solver proofs. In *Proceedings of the 5th Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, pages 26–40, July 2014. Available at https://easychair.org/publications/paper/N3GD.

[81] Christine Solnon. Benchmarks for the subgraph isomorphism problem. https://perso.citi-lab.fr/csolnon/SIP.html, 2019.

[82] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In *Proceedings of the 12th IAPR-TC-15 International Workshop on Graph-Based Representation in Pattern Recognition (GbRPR '19)*, pages 1–13, June 2019.

[83] Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. CrystalBall: Gazing in the black box of SAT solving. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *Lecture Notes in Computer Science*, pages 371–387. Springer, July 2019.

[84] Bernardo Subercaseaux, Wojciech Nawrocki, James Gallicchio, Cayden Codel, Mario Carneiro, and Marijn J. H. Heule. Formal verification of the empty hexagon number. In *Proceedings of the 15th International Conference on Interactive Theorem Proving (ITP '24)*, volume 309 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:19, September 2024.

[85] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, March-April 2021.

[86] Cesare Tinelli. Scalable proof production and checking in SMT. In *Proceedings of the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT '24)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:2, August 2024.

[87] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *10th International Symposium on Artificial Intelligence and Mathematics (ISAIM '08)*, 2008. Available at http://isaim2008.unl.edu/index.php?page=proceedings.

[88] Dieter Vandesande, Jordi Coll, and Bart Bogaerts. Certified branch-and-bound MaxSAT solving. In *Proceedings of the 40th AAAI Conference on Artificial Intelligence (AAAI '26)*, pages 14342–14351, January 2026.

[89] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. QMaxSATpb: A certified MaxSAT solver. In *Proceedings of the 16th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[90] VeriPB: Verifier for pseudo-Boolean proofs. https://veripb.org.

[91] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.