Certifying Combinatorial Optimization Using Pseudo-Boolean Reasoning

Andy Oertel

Licentiate Thesis, 2025

Department of Computer Science
Lund University

Funding information: Andy Oertel was funded by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

ISBN 978-91-8104-770-7 (electronic version)
ISBN 978-91-8104-769-1 (print version)
ISSN 1652-4691
Licentiate Thesis 6, 2025

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

E-mail: `andy.oertel@cs.lth.se`
WWW: `aoertel.de`

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2025

Abstract

Combinatorial optimization provides a powerful framework for solving complex optimization problems with general-purpose solvers by modelling the problem in an abstract language. Due to breakthroughs in algorithms to solve combinatorial optimization problems in last decades, combinatorial optimization has become a valid approach to solve many real world problems efficiently. Key application areas are planning, scheduling, computer-aided design, verification, and even theorem proving. However, the efficiency of the tools to solve these problems comes at the cost of increased complexity of the solvers. This makes it difficult to trust that the result computed by a combinatorial optimization solver is correct, which especially becomes a concern if the correctness of the result is mission-critical.

The main approach to address this issue is certifying algorithms, where an algorithm has to also generate a certificate that its result is correct, which can then be checked independently. This thesis demonstrates how pseudo-Boolean reasoning can be used to provide efficient certification of results returned by different kinds of combinatorial optimization solvers. We present a unified multipurpose certification system with a formally verified end-to-end verification toolchain, which guarantees that the combinatorial optimization problem was solved correctly. Developing a multipurpose certification system distinguishes this work from any prior work, which predominantly focused on very specialized approaches. We also present certification for many algorithms which, prior to our work, lacked any approach for certifying their result.

Contribution Statement

The following papers are included in this thesis:

Paper I Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version 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.

Paper II 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.

Paper III Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "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.

Paper IV Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. "Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs". In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

Paper V 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, Febuary 2024.

Paper VI 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.

There are additional papers that Andy Oertel contributed to, but they are not included in this thesis:

- Emir Demirović, Ciaran McCreesh, Matthew J. 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.

- Wietze Koops, Daniel Le Berre, Magnus O. Myreen, Jakob Nordström, Andy Oertel, Yong Kiam Tan, 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.

The table below indicates the responsibilities Andy Oertel had in writing the papers. For all papers, there were always ongoing discussion between all authors, especially at early stages of the project.

Paper	*Concept*	*Implementation*		*Evaluation*	*Writing*
		Verifier	*Solver*		
I	◑	○	◕	◔	◑
II	◕	●	◕	●	◕
III	◑	●	◔	●	◑
IV	◑	●	○	◑	◑
V	◑	◕	○	○	◔
VI	◑	◕	○	●	◑

The dark portion of the circle represents the amount of work and responsibilities assigned to Andy Oertel for each individual step:

● Andy Oertel led and did almost all the work.

◕ Andy Oertel led and did a majority of the work.

◑ Andy Oertel was a contributor to the work.

◔ Andy Oertel was a minor contributor to the work.

○ Andy Oertel did not contribute, except for proofreading.

Concept Coming up with the ideas and working out the details in theory.

Implementation Writing the software required for the paper.

Evaluation Conducting the experimental evaluation and analysing the data.

Writing Drafting and editing the paper.

The following discusses the contributions by Andy Oertel in more detail.

Paper I

Andy Oertel developed the theory for certifying the binary adder encoding, with regular discussions with Jakob Nordström, and helped Stephan Gocht in developing the general framework to certify different encodings. Stephan Gocht provided prototype implementations for the sequential counter encoding and the (generalized) totalizer encoding. The final implementation was done by Andy Oertel with the help of Ruben Martins who implemented the certifying sequential counter encoding. The benchmark set and data to be measured in the experiments was discussed with all authors. Andy Oertel wrote the binary adder section of the paper, provided figures and examples to all sections of the paper and improved the evaluation section. All authors were involved in discussions about the structure of the paper and helped to polish the manuscript.

Paper II

Andy Oertel was the lead author of the paper. All authors were involved in discussions to develop the theoretical foundations to certify core-guided MaxSAT solvers. Andy Oertel figured out the details to make certification feasible in practice. Andy Oertel and Dieter Vandesande implemented certification into CGSS. Dieter Vandesande implemented the certification for the incremental totalizer encodings and Andy Oertel implemented everything else. Andy Oertel implemented improvements in VeriPB improve the proof checking performance. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. Andy Oertel found the bug in the original implementation of CGSS and provided fixes for the bug. The structure of the paper was discussed with all authors. The preliminaries, the description of how to do certification for core-guided MaxSAT, and the experimental evaluation were written by Andy Oertel, which were later improved by all authors.

Paper III

The theoretical idea to certify the dynamic generalized polynomial watchdog encoding was developed by Bart Bogaerts. This idea was discussed and improved through discussions with all authors. Jeremias Berg, Bart Bogaerts, and Andy Oertel discussed alternative approaches and found good reasons that explain why shadow circuits are required. Tobias Paxian and Dieter Vandesande implemented certification in the MaxSAT solver Pacose with assistance from Andy Oertel. Improvements and additional tracking of proof statistics in the proof checker were implemented by Andy Oertel. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. The structure of the paper was discussed with all authors. Andy Oertel wrote the preliminaries on pseudo-Boolean proof logging and the experimental evaluation, which were later improved by all authors.

Paper IV

Most of the certification was developed by Ambros Gleixner, Alexander Hoen, and Jakob Nordström, while Andy Oertel helped to figure out certification for some remaining presolving techniques. The need for an objective update rule was discovered by Alexander Hoen and Andy Oertel. Andy Oertel developed the objective update rule and implemented it together with further checking improvements into the proof checker VeriPB. Alexander Hoen implemented certification in the MIP presolver PaPILO. Alexander Hoen and Andy Oertel jointly conducted the experiments and Alexander Hoen analysed the experimental data. The structure of the paper was discussed with all authors. Andy Oertel wrote the preliminaries on pseudo-Boolean proof logging and about the new objective update rule, and he helped Alexander Hoen in writing the description on the certification of presolving reductions.

Paper V

Andy Oertel joined this project after it was running and only the other authors contributed to the original idea of the project. Andy Oertel developed all elaboration algorithms, which were discussed in meeting with Jakob Nordström. Andy Oertel improved the elaboration algorithm to achieve the necessary performance. The prototype implementation of the elaboration in VeriPB was done by Stephan Gocht. Andy Oertel changed major parts of this preliminary implementation to accommodate further improvements to the elaboration algorithms and support for all proof rules. Andy Oertel also implemented checked deletion into VeriPB. Magnus Myreen and Yong Kiam Tan implemented the formally verified proof checker CakePB, where Andy Oertel helped to formalize the correctness proof for strengthening rules. The structure of the paper was discussed with all authors. Andy Oertel wrote the proof elaboration section, which was later improved by all authors.

Paper VI

Certification for almost all MaxSAT preprocessing techniques was developed by Jeremias Berg, Hannes Ihalainen, and Matti Järvisalo. Jakob Nordström and Andy Oertel helped them to develop certification for the techniques of hardening and label matching. The proof format for problem reformulation proofs was jointly developed through discussions with all authors. Hannes Ihalainen implemented the certification into the MaxSAT preprocessor MaxPre. Support for problem reformulation was implemented by Andy Oertel into VeriPB and by Magnus Myreen and Yong Kiam Tan into CakePB. Andy Oertel conducted the experiments and analysed the experimental results. All authors were involved in discussing benchmark sets and data to be measured. The structure of the paper was discussed with all authors. Andy Oertel wrote the preliminaries and the experimental evaluation, which was later improved by all authors.

Acknowledgements

I would like to take this opportunity to thank my family for always supporting me on my educational journey. Without my parents, sister, and brother I would not have made it this far. I especially also want to thank Johanna for all the time we spent together and will spend together. Thank you for always encouraging me and your unwavering support. Martin, Max, and Sebastian, I am very happy to have you as my friends and enjoy every moment we spend together.

When it comes to the work environment, many thanks to Benjamin, Christophe, David, Duri, Gaia, Jonas, Kilian, Morgan, Noel, Rui, Shuo, Stefan, Wietze, and Yassine for your company in and outside of office. I would also like to thank the people in the CP, OR, and SAT communities for welcoming me. I am happy to be friends with many of you.

I would also like to thank the administration at the Department of Computer Science of LTH for always being very helpful. Also, thank you, Mikkel and Anne for creating such a lovely research environment in Copenhagen. I always enjoyed being at BARC and joining the events you organized.

I would like to thank all my co-authors and everyone who worked on VeriPB for helping me with this endeavour. I would not have been able to do all this work I am presenting in this thesis without you. The discussions with all of you were always fruitful and enlightening. Especially, I would like to thank Bart Bogaerts, Stefan Gocht, Ciaran McCreesh, and Jakob Nordström for designing the original VeriPB proof system, which my work builds upon.

I would also like to thank the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation for funding my research and my stay in Berkeley. Additionally, I would like to thank the SAT association, Zuse Institute Berlin, Landshövding Per Westlings Minnesfond, COST action *EuroProofNet*, and CADE conference for providing me with additional travel funding. For hosting the computational resources to run my experiments, I would like to thank LUNARC at Lund University.

Last but not least, I thank my PhD supervisor Jakob Nordström for introducing me to the topic, proposing all kinds of ideas, and making all of this possible. I also thank my co-supervisor Susanna F. de Rezende for supporting me during my PhD.

Thank you very much!

Tack så mycket!

Vielen Dank!

Contents

Certifying Combinatorial Optimization Using Pseudo-Boolean Reasoning

1 Introduction

In the last couple of decades, combinatorial optimization solvers have been vastly improved across all paradigms and are able to solve very large optimization problems efficiently. This revolution led to combinatorial optimization solvers of all paradigms being used for many commercial and academic applications. For instance, Boolean satisfiability (SAT) and maximum Boolean satisfiability (MaxSAT) solving [BHvMW21] are used for, e.g., hardware verification [BCCZ99], in chip design [CNR21], or to prove theorems [HKM16, SH23]. Constraint programming (CP) [RvBW06] is used for solving, e.g., personnel allocation and timetabling [Wal96], power plant production planning [BBVC13], and sports league scheduling [Wei25]. Mixed integer programming (MIP) [AW13] is used for, e.g., supply chain optimization [GGK$^+$19], public transport planning [S$^+$20], and investment portfolio optimization [MOS15].

However, the improved performance of solvers comes at the cost of complexity by using more sophisticated and specialized reasoning techniques. This complexity naturally raises the question if modern combinatorial solvers are implemented correctly. Correctness is especially crucial when combinatorial optimization solvers are used to solve mission-critical problems, like for ambulance dispatch [Sch12], kidney exchange programs [MO12], or air traffic control [HPRS24]. Specifically, it well-know that combinatorial optimization solvers for all paradigms and of different levels of maturity contain bugs and can return incorrect results [BLB10, CKSW13, AGJ$^+$18, GSD19, GS19, BBN$^+$23, PB23, WS24]. Since mature solvers where many people reviewed the code contain bugs, it is even harder to believe that new cutting edge techniques are implemented correctly. To mitigate this issue the following approaches are known in software engineering.

The most used approach in software engineering is *testing*, where a program is checked to give the correct output given a specific input [MSB11]. However, this approach is limited to known pairs of input and output, which can be generated manually or automatically, e.g., by *fuzzing* [MKL$^+$95, ZWCX22, PB23]. Considering the number of found bugs, the most successful approach for testing in combinatorial optimization is fuzzing with a structured way to generate instances that triggers the use of all possible techniques and their interaction in the solver [ABS13, PB23]. To conclude, testing can only show the existence of a fault, but there are no guarantees on the correctness of the software.

The other extreme is *formal verification* of software by formally specifying the behaviour of the program and proving that the implementation adheres to this specification [HT15]. This approach fully guarantees that the software adheres to the specification, but the effort involved in formally verifying a program is huge and does not scale to the size and complexity of modern solvers. The most advanced formally verified solver is the SAT solver IsaSAT [FL23], which performs significantly worse than any other modern solver and SAT solving is the least complex combinatorial optimization paradigm.

The approach of *certifying algorithms* [MMNS11], which we will focus on in this thesis, provides a good middle ground between testing and formal verification.

The idea of certifying algorithms is that an algorithm not just returns an output, but also a certificate that shows that the output is correct. Using the certificate, it should be easy to verify that the output is correct for the given input to the algorithm. Hence, with certifying algorithms we do not need to trust the implementation or algorithm to trust that the output is correct. We only need to trust a simpler algorithm that checks that the output is correct using the certificate, which is typically simple enough so that this implementation can be formally verified to be correct. Therefore, we obtain the guarantee that the output is correct for the given input, which is actually the main guarantee that is usually interesting, since we just want to know that our problem was solved correctly.

While certifying algorithms existed already in the form of the extended Euclidean algorithm [MMNS11] and primal-dual optimization algorithms [Far02], on a large scale certifying algorithms were first explored in the LEDA project[MN89, MN95]. The term certifying algorithms was first used by Kratsch et al. [KMMS06]. In combinatorial optimization, certifying algorithms are becoming increasingly popular in many paradigm [BFT11, CGS17, VS10]. Especially in the community of SAT solving, certifying algorithms became so mainstream that since 2013 all solvers competing in the main track of the annual SAT competition require certification. This interest and the proximity to the theory of proof complexity [BN21] led to many certification systems for SAT solving [Heu21], like RUP [GN03, Van08], DRAT [JHB12], FRAT [BCH21], PR [HKB17], and SR [BT21].

In this thesis, we are studying and extend a certification system based on pseudo-Boolean reasoning called VeriPB [BGMN23, GN21, Goc22]. VeriPB is inspired by the success of certification in SAT solving and is based on the cutting planes proof system [CCT87] from the theory of proof complexity [BN21]. This thesis extends the VeriPB system from the certification of decision problems to optimization problems by introducing new rules that capture the reasoning in optimization solvers and making it possible to certify bounds on the optimal value. This thesis also shows how to certify problem reformulations independent of solving using VeriPB, where it is possible to certify various guarantees on the reformulated problem in relation to the original problem. Finally, to provide formal guarantees that output is correct, we develop a framework for formally verified certificate checkers called CakePB, which makes it easy to get formally verified checkers for different combinatorial optimization paradigms.

This thesis makes progress towards the development of a general certification framework for all combinatorial optimization paradigms using one unified multipurpose system. Specifically, this thesis shows that VeriPB can be used to obtain certification for various algorithms to solve MaxSAT, subgraph solving algorithms, and preprocessing and presolving techniques for MaxSAT and 0-1 integer linear programming. Additionally, in related work it has been shown how to use VeriPB for certification of advanced SAT solving techniques [GN21, BGMN23], dynamic programming algorithms [DMM^{+}24], constraint programming solvers [EGMN20, GMN22, MM23, MMN24, MM25], automated planning [DHN^{+}25], and pseudo-Boolean optimization [KLM^{+}25]. It has also been proposed to extend the VeriPB certification system to fully support certification of mixed integer programming [DEGH23], which would directly enable certification for any kind of combinatorial optimization.

The first part of this thesis is a comprehensive summary of the work (the so-called *kappa*), which is structured as follows. In Section 2, background for the topics discussed in this thesis is introduced. This includes a review of different combinatorial optimization paradigms and an introduction to certifying algorithm. In Section 3, related work is reviewed that also studies certifying algorithms for combinatorial optimization. The pseudo-Boolean certification system that we study to certify combinatorial optimization algorithms is presented in Section 4 and is a full description of the certification system including all contributions by this thesis. In Section 5, the contributions of this thesis are discussed in detail. The introduction of the thesis ends with some concluding remarks and future work in Section 6. The second part of this thesis consists of included papers.

2 Background

This section introduces the necessary preliminaries required to understand this thesis together with the used notation. It is assumed the reader has basic knowledge in Theoretical Computer Science, including basic computational complexity, logic, and graph theory. For additional background on computational complexity and logic see [AB16] and for some background on graph theory see [Die16].

2.1 Basic Notation

It follows some review of standard logic notation, which can be found, e.g., in [AB16]. We use $\top$ to denote true (tautology) and $\bot$ to denote false (contradiction). The symbol $\wedge$ denotes a logical conjunction, $\vee$ a logical disjunction, $\Rightarrow$ a material implication, and $\Leftrightarrow$ a material equivalence. A *Boolean variable* is a variable with domain $\{\bot, \top\}$, where often $\bot$ is associated with 0 and $\top$ with 1. A *literal* of a Boolean variable x is either the Boolean variable itself x or its *negation* $\overline{x}$ or also $\neg x$.

2.2 Combinatorial Optimization

We will start this section with a review of standard notation and definitions that will be used throughout this thesis, which can be found in, e.g., [Sau24, BN21]. For more history on combinatorial optimization, see [Sch05].

The goal of *mathematical optimization* is to find an optimal element in a set of feasible elements [Sau24]. A *combinatorial optimization* problem is a special case of mathematical optimization problem where the feasible elements are (at least partially) from a discrete set. While most of the definitions can be extended to arbitrary sets, in this thesis we only consider *Boolean optimization problems* (aka 0–1 *optimization problems*), i.e., the feasible elements are from a subset of $\{0, 1\}^n$. The optimal value is usually defined by an *objective* function mapping an element from the set to a numerical value that without loss of generality should be minimized, hence for this thesis the objective function $f : \{0,1\}^n \rightarrow \mathbb{Z}$ should be minimized. A *constraint* to be a function $C : \{0,1\}^n \rightarrow \{\bot, \top\}$. The set of feasible elements is further restricted by a set of constraints so that an element is feasible if and only if all constraints evaluated on this element return $\top$, i.e., the conjunction of all constraints evaluates to $\top$. A *trivial constraint* maps to $\top$ for any input and a *contradictory constraint* maps to $\bot$ for any input.

A *decision problem* is a special case of an optimization problem where we are only interested in knowing if there is a feasible element with respect to the constraint. Hence, the objective function can be viewed as being constant, e.g., $f : \vec{x} \mapsto 0$.

Each dimension of the feasible set $\{0,1\}^n$ is associated with a Boolean variable. For a set of constraint F or an objective f, we use the notation $F(\vec{x})$ or $f(\vec{x})$ to stress that F or f is defined over the vector of Boolean variables $\vec{x} = x_1, \ldots, x_n$, respectively. We syntactically highlight a partitioning of the vector of Boolean variables by writing $F(\vec{y}, \vec{z})$ or $F(\vec{a}, \vec{b}, \vec{c})$ meaning $\vec{x} = \vec{y}, \vec{z}$ or $\vec{x} = \vec{a}, \vec{b}, \vec{c}$, respectively.

A *(partial) assignment* ρ is a (partial) function from variables to $\{\bot, \top\}$. A *substitution* ω is a generalization of an assignment by allowing variables to map to literals. Hence, we consider a (partial) assignment to be a special case of a substitution, where all unassigned variables map to themselves. Substitutions are extended to literals by defining for the negation of a variable that $\omega(\overline{x}) = \neg\omega(x)$, and to preserve truth values, i.e., $\omega(0) = 0$ and $\omega(1) = 1$. When denoting a substitution, then all variables that are not explicitly mentioned are mapped to themselves, e.g., the substitution $\{x \mapsto \overline{y}, z \mapsto 0\}$ maps x to $\overline{y}$, z to 0, and all other variables to themselves.

For a list of variables $\vec{x} = x_1, \ldots, x_n$ and a substitution ω, we define that $\omega(\vec{x}) = \omega(x_1), \ldots, \omega(x_n)$. A substitution α can be *composed* with another substitution ω by applying ω first and then α, i.e., $(\alpha \circ \omega)(\vec{x}) = \alpha(\omega(\vec{x}))$. We can *apply a substitution* ω to a constraint $C(\vec{x})$, which is denoted by $C(\vec{x})\restriction_\omega$ or $C\restriction_\omega$, by first applying ω on $\vec{x}$ and then evaluating C on $\omega(\vec{x})$, i.e., every variable x_i of C is substituted by $\omega(x_i)$. A substitution ω *satisfies* a constraint C if $C\restriction_\omega$ is trivial and *falsifies* C if $C\restriction_\omega$ is contradictory.

There are many paradigms of combinatorial optimization that have been studied, where each paradigm restricts the combinatorial optimization problem in some way or takes a different view on how the constraints are formulated. Some key paradigms that are relevant for this thesis are introduced in the rest of this section.

2.2.1 Boolean Satisfiability (SAT)

The *Boolean satisfiability (SAT) problem* is one of the core decision problems in computer science [BHvMW21] and is the canonical NP-complete problem [Coo71, Lev73]. To define the SAT problem we need some further notation. A *(disjunctive) clause* is a logical disjunction of literal, e.g., $x \lor \overline{y} \lor z$, which is the type of constraint considered for the SAT problem. Without loss of generality, a *Boolean formula* is in *conjunctive normal form (CNF)*, which is a conjunction of clauses, e.g., $(x \lor \overline{y}) \land (\overline{x} \lor y) \land (\overline{x} \lor \overline{y})$.

The SAT problem is a decision problem that asks if there exists an assignment that satisfies a Boolean formula. If there exists such an assignment, then the formula is said to be *satisfiable*. If there does not exist such an assignment, then the formula is said to be *unsatisfiable*.

The SAT problem can also be phrased in terms of a combinatorial decision problem. An element for in a SAT problem is an assignment, the constraints are the clauses, and we want to find an assignment that satisfies all clauses. We can also additionally associate the truth value $\bot$ with 0 and $\top$ with 1 so that the considered elements are in $\{0, 1\}^n$.

At the core of all modern SAT solving algorithms *conflict-driven clause learning (CDCL)* is used, which is enhanced with techniques for pre- and inprocessing, which are discussed in Section 2.2.4. We will briefly review the CDCL algorithm presented in Algorithm 1, where more detail about the components can be found in [MSLM21].

One subroutine of the CDCL algorithm is *unit propagation*. Given a partial assignment ρ, a clause C unit propagates the literal ℓ if all literals except ℓ are mapped to $\bot$ by ρ. Then the resulting assignment is $\rho \circ \{\ell \mapsto \top\}$. The first step in the CDCL loop is to do unit propagation on the clauses in F starting with the partial assignment ρ. If the current assignment ρ satisfies the formula, then the formula is satisfiable. Otherwise, we check if there is a clause that is falsified by ρ. If there is no such clause, we assign an unassigned variable to a value. Otherwise, we learn a new clause based on the propagations and decisions that were responsible to falsify the clause, which is called *conflict analysis*. If the learnt clause is the empty clause, then we know that the formula is unsatisfiable, as the empty clause cannot be satisfied by any assignment. Otherwise, the learnt clause is added to the formula and some variables are unassigned, which is called *backjumping*.

2.2.2 Maximum Satisfiability

The canonical extension of the SAT problem to an optimization problem is the *maximum satisfiability (MaxSAT)* problem [BJM21]. Given a set of (weighted) soft clauses and a set of hard clauses, the MaxSAT problem asks for an assignment that maximizes the sum of the weights of satisfied soft clauses subject to satisfying all hard clauses. In practice, the MaxSAT problem is more commonly formulated as finding an assignment that minimizes an integer linear function over literals f subject to satisfying all clauses, where $\bot$ and $\top$ are associated with 0 and 1, respectively. Without loss of generality, all coefficients in the objective function f are assumed to be positive by using that $\overline{x} = 1 - x$. Hence, we will assume for the rest of the thesis that a MaxSAT problem is given in the latter formulation.

These two formulations can be translated into each other such that each solution to one problem is also a solution to the other problem with the same objective function value. The translation from the second to the first formulation is straightforward by considering all constraints as hard clauses and for each term in the objective function the negated literal is added as a soft clause weighted by the coefficient of the term. The negated literal changes the problem from a

Algorithm 1: Basic skeleton of the conflict-driven clause learning algorithm, which is the core algorithm of modern SAT solvers. The input is a Boolean formula F in CNF and the output is if F is satisfiable or not.

```
conflictDrivenClauseLearning($F$):
$\rho \leftarrow \emptyset$;
Loop
    $\rho \leftarrow$ unitPropagation($F, \rho$);
    if $\rho$ satisfies $F$ then
        return SAT;
    if $\rho$ falsifies a clause in $F$ then
        $C \leftarrow$ conflictAnalysis($F, \rho$);
        if $C$ is the empty clause then
            return UNSAT;
        $F \leftarrow F \cup C$;
        $\rho \leftarrow$ backjump();
    else
        $\rho \leftarrow$ decideVariable();
```

minimization problem to a maximization problem. To translate the first to the second formulation, we add a new variable b_i for each soft clause C_i with weight w_i and add $b_i \vee C_i$ to the hard clause. Then the resulting hard clauses are the constraints and the objective is $\sum_i w_i b_i$.

There are several state-of-the-art solving techniques for MaxSAT that are based on SAT solvers. The most straightforward algorithm to solve MaxSAT is *solution-improving search (SIS)* [ES06, PRB18], which is outlined in Algorithm 2. The idea is to solve the constraints using a SAT solver. If there is a solution, then we compute the objective value for this solution and add clauses to the constraints which enforce to find a solution that has a strictly better objective value. Then we repeat to solve this problem with a SAT solver until the SAT solver returns that the constraints is unsatisfiable, which means that the best solution found so far is the optimal value. There are various encodings known to enforce a strictly better solution [War98, BB03, ES06, JMM15, PRB18]. An *incremental SAT solvers* [ES03] reuses information from previous calls to the solver and can be called with so-called *assumptions*, which is a partial assignment that should be extended to a full assignment by the solver. Such a solver helps to speed up SIS MaxSAT solvers and allows efficient encodings that only change the used assumptions from one call to the next.

Incremental SAT solvers also make *core-guided* MaxSAT solvers [MDM14, IBJ21] possible. Another feature of incremental SAT solvers is that if the assumptions cannot be extended to a complete solution, then the solver returns a subset of the variables assigned in the assumptions where at least one variable should be set

Algorithm 2: Basic skeleton of the solution-improving search algorithm to solve the MaxSAT problem with objective f and Boolean formula F in CNF. The output is the optimal value of the MaxSAT problem, where the optimal value ∞ means that F is unsatisfiable.

```
solutionImprovingSearch($F, f$):
$v \leftarrow \infty$;
Loop
    $(sat?, \rho) \leftarrow$ solveSAT($F$);
    if $sat? = UNSAT$ then
        return $v$;
    $v \leftarrow f(\rho)$;
    $F \leftarrow F \cup$ asCNF($f \leq v - 1$);
```

to the opposite value to satisfy the formula. This subset $\{\ell_1, \dots, \ell_n\}$ is a so-called *core* and expresses a clause $\ell_1 \vee \cdots \vee \ell_n$ that is satisfied if at least on variable is assigned to the opposite value than used in the assumptions. We will focus on the state-of-the-art OLL algorithm [AKMS12, MDM14] to handle the core clause, but there are other algorithms like PMRes [NB14] that differentiate on how they treat the core clause. A general skeleton of the core-guided algorithm to solve MaxSAT is outlined in Algorithm 3, where we use lits(f) to denote the set of literals in the objective.

The OLL algorithm first calls the SAT solver with the assumptions that set every literal in the objective to $\bot$, which is the partial assignment that leads to the smallest possible objective value. If SAT solver is able to extend these assumptions to a complete assignment that satisfies all constraints, then we found an optimal solution, as the assumptions enforce the smallest possible objective value that can be achieved with this objective. If the SAT solver is not able to do this, it will return a core clause C. We say that the *weight* $w(C, f)$ of a core C is the smallest coefficient of a literal in C in the objective f. We will introduce as many new variables $c_1, \dots, c_n$ as there are literals in C and add clauses enforcing that $c_i \Leftrightarrow \sum_i \ell_i \geq i$ for $i = 1, \dots, n$, i.e., c_i is true if at least i literals of C are true. There is now an equivalence between ℓ_i literals and c_i variables, so that $\sum_i \ell_i = \sum_i c_i$, which can be used to substitute the expression $\sum_i w(C, f)\ell_i$ in f by $\sum_i w(C, f)c_i$ resulting in the reformulated objective f_{ref}. This process is then repeated with the reformulated problem.

There are many other approaches used to solve MaxSAT, which are not of interest for this thesis. Additional approaches to solve MaxSAT using incremental SAT solvers are *implicit hitting set (IHS)* search [DB13] or *branch and bound MaxSAT* solvers [AH14, LXC+21]. Furthermore, there are approaches that do not rely on SAT solvers at all to solve the MaxSAT problem like *integer-linear programming (ILP) solvers* [Ach07].

Algorithm 3: Basic skeleton of the core-guided algorithm for solving the MaxSAT problem with objective f and Boolean formula F in CNF. The output is the optimal value of the MaxSAT problem, where the optimal value ∞ means that F is unsatisfiable.

```
coreGuidedSearch($F, f$):
Loop
    $\alpha \leftarrow \{\ell \mapsto 0 | \ell \in \text{lits}(f)\}$;
    $(sat?, \rho, \kappa) \leftarrow$ solveWithAssumptionsSAT($F, \alpha$);
    if $sat? = UNSAT$ then
        if $\kappa = \emptyset$ then
            return $\infty$;
        $(F, f) \leftarrow$ reformulateProblem($F, f, \kappa$);
    else
        return $f(\rho)$;
```

2.2.3 Pseudo-Boolean Optimization

A further generalization of SAT and MaxSAT is *pseudo-Boolean optimization (PBO)* problem [RM21]. In pseudo-Boolean optimization the constraints are *pseudo-Boolean (PB) constraints*, which are integer-linear inequalities over literals and the objective function is an integer linear function over literals. Here we are again using that convention that $\bot$ and $\top$ are associated to 0 and 1, respectively, and that the negation $\overline{x} = 1 - x$. Without loss of generality, these constraints are in normalized form $\sum_i a_i \ell_i \geq A$, where the coefficients a_i and the right-hand side A are non-negative integers and the literals ℓ_i are over distinct variables. The right-hand side A is also referred to as the *degree (of falsity)*. We use $\doteq$ to denote *syntactic equivalence* and to avoid confusion with the operator $=$. SAT is a special case of PB solving, as a clause $\bigvee_i \ell_i$ has the same semantics as the pseudo-Boolean constraint $\sum_i \ell_i \geq 1$.

Pseudo-Boolean optimization is equivalent to 0–1 ILP, where negative literals are turned into positive literals using that $\overline{x} = 1 - x$. Hence, any ILP solver [Ach07] can be used to solve PBO. However, there are also specialized PB solvers that follow the idea of CDCL presented in Algorithm 1, which gives rise to solve the PB decision problem [LP10, EN18]. Pseudo-Boolean optimization can then be solved using the MaxSAT approaches discussed in Section 2.2.2 using an incremental PB decision solver instead of a SAT solver [DGD+21].

The only components that need to be changed to use Algorithm 1 to solve a pseudo-Boolean problem are the procedures for unit propagation, conflict analysis, and backjumping. There is a lot of literature about pseudo-Boolean conflict analysis [LP10, EN18], but they are generally more complex as conflict analysis in SAT solvers, but possible. To see that we can still efficiently do unit propagation with pseudo-Boolean constraints we need to define the *slack* of a

PB constraint, which measures how close a constraint is to be falsified by an assignment. The slack of a constraint $C \doteq \sum_i a_i \ell_i \geq A$ under the assignment ρ is $\mathit{slack}(C, \rho) \coloneqq \sum_{\rho(\ell_i) \neq 0} a_i - A$. Hence, if $\mathit{slack}(C, \rho) < 0$, then C is falsified by ρ, as even setting all literal in C that are unassigned by ρ to 1 would not satisfy C. A PB constraint C containing $a_i \ell_i$ as a term propagates ℓ_i to 1 under an assignment ρ if and only if $\mathit{slack}(C, \rho) < a_i$, as setting ℓ_i to 0 would result in $\mathit{slack}(C, \rho \circ \{\ell_i \mapsto 0\}) < 0$, which says that C is falsified by $\rho \circ \{\ell_i \mapsto 0\}$. Efficient propagation is an active research area [Dev20b, NORZ24].

Another approach to solver pseudo-Boolean optimization problems is to encode the PB constraints into clauses and then use a SAT solver to solve the constraints [ES06, MML14, SN15]. There are many encodings with different properties that are used to encode PB constraints into CNF [Bat68, War98, BB03, ES06, JMM15, PRB18]. These solvers use standard SAT solvers and directly benefit from any improvement for SAT solving.

2.2.4 Preprocessing

When solving combinatorial optimization problems in practice, many solvers first use some algorithm to reformulate the problem before using the main solving procedure. This approach of reformulating the problem is called *preprocessing*, which is sometimes also referred to as *presolving*. It is also possible to reformulate the current problem maintained by the solver during the main solving procedure, which is called *inprocessing*. It has been shown for all kinds of combinatorial optimization paradigms that preprocessing is an important technique [ABG+20, IBJ22, HGH23]. This section only gives a brief overview over preprocessing and will not go into detail about specific techniques used in preprocessing. Specific preprocessing techniques are discussed in [ABG+20, BJK21, IBJ22].

Preprocessing techniques can be grouped into two categories. The first category are so-called *primal preprocessing* techniques, which preserve the set of feasible solution and only change how this set is described. The second category of techniques can change the feasible set if it is guaranteed that the optimal value stays unchanged, which are called *dual preprocessing* techniques. A special kind of dual preprocessing techniques is *symmetry breaking*, which restricts the set of feasible solutions to only contain a few feasible solutions from the symmetric set of solutions by introducing new constraints. Symmetry breaking is commonly only preformed syntactically over the constraints describing the feasible set.

2.3 Proof Complexity Proof Systems

The research area of *proof complexity* studies how efficiently reasoning systems can prove statements [Kra19]. A key concept of proof complexity is the so-called proof system and was defined by Cook and Reckhow [CR79]. This thesis only considers *sequential refutation proof systems*, hence we will just refer to them as *proof systems* in this thesis.

The following definition of a (refutation) proof system is presented in the language of combinatorial optimization, but proof systems are commonly only used for decision problems. A refutation proof system is a set of inference rules to derive new constraints. A *proof* is a sequence of inference rule applications that start with the original constraints describing the set of feasible solutions and each inference rule application adds a new constraint. A proof system must satisfy the following three conditions to be a proof system in the sense of Cook and Reckhow [CR79]:

Soundness: If there is a feasible solution, then the proof system can not show that there is no feasible solution.

Completeness: If there is no feasible solution, then there exists a proof in the proof system showing that there is no feasible solution.

Polynomial time checkable: Each inference rule application can be checked in polynomial time in the size of the size of the proof.

All inference rules that we consider in this section are polynomial time checkable, since syntactic checks of the hypotheses are sufficient. By using the soundness property, we can show that there were no feasible solutions for a problem if we can derive a constraint that obviously states that is a contradictory constraint.

To denote the inference rules in a proof system, we will use the notation

$$\frac{H_1 \qquad \dots \qquad H_n}{C}\ \text{Inference rule}$$

to say that the *conclusion* C can be derived if the *hypotheses* $H_1, \dots, H_n$ have been derived before or are part of the original formula. We will now introduce a few proof systems that are relevant for this thesis.

In the *resolution* proof system [Bla37, DP60, DLL62, Rob65] operates over clauses, i.e., the conclusion and hypotheses are all clauses. This means that if we manage to derive the empty clause, then this shows that the original set of clauses was unsatisfiable. The only inference rule in the resolution proof system is the *resolution rule*

$$\frac{C \vee x \qquad D \vee \overline{x}}{C \vee D}\ \text{Resolution over } x$$

deriving the clause $C \vee D$ from the clauses $C \vee x$ and $D \vee \overline{x}$. This rule is sound, since any assignment satisfying $C \vee x$ and $D \vee \overline{x}$ has to satisfy C if $x = \bot$ or D if $x = \top$. For a proof that the resolution rule is complete, see [Rob65].

The *cutting planes* proof system [CCT87] uses pseudo-Boolean constraints. This means that if we derive the constraint $0 \geq 1$, then the original set of pseudo-Boolean constraints was unsatisfiable. For additional details with of the cutting planes proof system, we refer the reader to [BN21]. We will use the notation $F \vdash C$ to say that there is a cutting planes derivation from the original PB constraint F to derive the PB constraint C, and write $F \vdash F'$ if $F \vdash D$ for each $D \in F'$. We start with *literal axiom rule* for any literal ℓ

$$\frac{}{\ell \geq 0}\text{ Literal axiom for } \ell\,,$$

which states that the constraint $\ell \geq 0$ can always be derived. This rule is sound, since adding $\ell \geq 0$ is the same as all variables are between 0 and 1, hence every literal is at least 0.

Two pseudo-Boolean constraints can be added together using the *addition rule*

$$\frac{\sum_i a_i \ell_i \geq A \qquad \sum_i b_i \ell_i \geq B}{\sum_i (a_i + b_i)\ell_i \geq A + B}\text{ Addition}\,.$$

The addition rule is sound, since when we do not normalize the constraint, then the sums of the coefficients of the satisfied literals by an assignment ρ satisfying both constraints $\sum_{\rho(\ell_i)=1} a_i$ and $\sum_{\rho(\ell_i)=1} b_i$ are larger than A and B, respectively. Hence, ρ also satisfies the sum of the constraints, as $\sum_{\rho(\ell_i)=1}(a_i + b_i)$ is larger than $A + B$.

A pseudo-Boolean constraint can be multiplied by a positive integer using the *multiplication rule*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i m a_i \ell_i \geq mA}\text{ Multiplication by } m \in \mathbb{N}\,.$$

Multiplication is sound, as the sum of the coefficients of the satisfied literals by an assignment is just multiplied by m and the degree is also just multiplied by m. Hence, any assignment satisfying the original constraint also satisfies the multiplied constraint.

A pseudo-Boolean constraint in normalized form can also be divided by a positive integer d if all coefficients are divisible by d using the *specialized division rule*

$$\frac{\sum_i d a_i \ell_i \geq A}{\sum_i a_i \ell_i \geq \lceil A/d \rceil}\text{ Division of normalized constraint by } d \in \mathbb{N}\,.$$

The division rule is sound, as the sum of the coefficients of the satisfied literals by an assignment is divided by d and so is the degree divided by d without considering rounding up. However, since all coefficients are integer, the satisfiability of the constraint does not change if the degree is rounded to the next biggest integer. By adding literal axioms before the division, division can be defined for all constraints without the condition that the coefficients are divisible. This yields the *(general) division rule*

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil}\text{ Division of normalized constraint by } d \in \mathbb{N}\,.$$

The rules so far are already sufficient to get a cutting planes proof system that is complete, since it can simulate resolution rule by adding the two hypotheses together and dividing by 2. However, Gocht et al. [GNY19] showed that adding the *saturation rule* [DG02] for normalized constraints

$$\frac{\sum_i a_i \ell_i \geq A}{\sum_i \min\{a_i, A\} \ell_i \geq A} \text{ Saturation of normalized constraint}$$

to the proof system yields a stronger proof system. This rule is sound, as the constraint is satisfied if a literal with a coefficient larger than A is satisfied and also if the coefficient is A.

Finally, we discuss *extension rules* that allow to derive constraints with new variables. The resolution proof system can be modified to the *extended resolution* proof system [Tse68] by adding the extension rule

$$\frac{q \notin \mathit{Vars}(F)}{\overline{\ell_1} \vee q \quad \overline{\ell_2} \vee q \quad \ell_1 \vee \ell_2 \vee \overline{q}} \text{ Extended resolution for new variable } q \text{ ,}$$

where $\mathit{Vars}(F)$ is the set of variables used by the original and derived constraints before this rule is applied. The extension rule introduces a new *extension variable* q and clauses forcing q to be true if and only if $\ell_1 \vee \ell_2$ is true. Similarly, *extended cutting planes* can be defined using the same rule.

2.4 Certifying Algorithms

This section briefly motivates and defines certifying algorithms. For more history and details on certifying algorithms see [MMNS11]. They also provide more examples of certifying algorithms for different types of problems.

The problem that certifying algorithms is trying to solve is to know if a software program is correct.[1] The approach of certifying algorithms tries to provide a middle ground between testing [MSB11] and formal verification [HT15] by providing correctness guarantees for the specific input we consider, but with less effort than what is required for formal verification. The idea of certifying algorithms is something we are all familiar with from solving school maths problems, e.g., solving equations. Then a way to make sure that the calculated result is correct would be to plug in the calculated values for the variables in the equations and check if all equalities hold. I.e., there is a procedure independent of the solving process to check the correctness of the result. This intuition can be formalized into certifying algorithms.

Before we can define what a certifying algorithm is, we require the following two definitions. The *precondition* of a function is the restriction on the input for which the function is valid. E.g., consider the function $\mathit{div}(a, b)$ for $a, b \in \mathbb{R}$ which divides a by b, then the precondition of $\mathit{div}(a, b)$ would be that $b \neq 0$, as division by 0 is undefined. The precondition can also be trivially satisfied, if the function is

[1]We consider a software program to be correct if it returns the correct output for the given input with respect to a formal specification of the function computed by the program. This means that an error in the specification is not considered incorrect for our purpose.

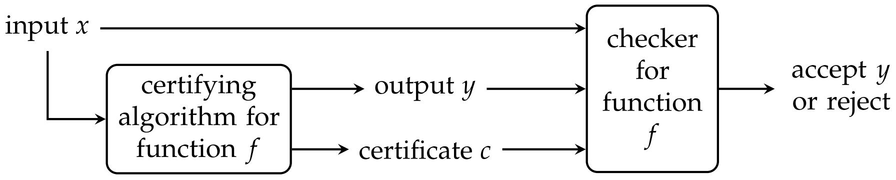


Figure 1: *Workflow for a certifying algorithm for the function f and checking the certificate with a checker for the function f.*

defined for any value of the function domain. The *postcondition* of a function is the expected output of the function with respect to the input of the function. E.g., for the function $div(a, b)$ the postcondition would be that value of the function $div(a, b)$ is actually a/b, which be verified by computing $a = b \cdot c$ where c is the computed result of $div(a, b)$.

A certifying algorithm get an input $x \in X$ and returns an output $y \in Y$ and a *certificate*[2] c. The x, y, and c are the input to the checker, which either verifies that the y is the correct output for x or the checker fails to verify the correctness of y. The latter case can either be due to y being the incorrect output or the certificate c does not show the correctness of y. It can be the case that the certificate is trivial (empty) if the checker can verify the correctness of output y for input x without additional information. See Figure 1 for the workflow of a certifying algorithm and the verification of the certificate.

In [MMNS11] 3 categories of certifying algorithms are defined. All definitions consider algorithms that compute the function $f : X \rightarrow Y$. *Strongly certifying algorithms* halt for all inputs $x \in X$ and the algorithm either returns that x does not satisfy the precondition and the certificate also shows that or returns $y \in Y$ and the certificate shows that $f(x) = y$. *(Ordinary) certifying algorithms* halt for all inputs $x \in X$ and the algorithm either returns that x does not satisfy the precondition and the certificate also shows that or returns $y \in Y$ and the certificate shows that $f(x) = y$ if x satisfies the precondition. In the latter case it can happen that if x does not satisfy the precondition, then $f(x) \neq y$. *Weakly certifying algorithms* only has to halt for x satisfying the preconditions. If the algorithm halts, then it either returns that x does not satisfy the precondition and the certificate also shows that or return $y \in Y$ and the certificate shows that $f(x) = y$ if x satisfies the precondition. Hence, if a weakly certifying algorithm halts, then it behaves exactly as an ordinary certifying algorithm.

To illustrate these different categories, we consider the above example of division. A strongly certifying algorithm for this problem would always halt and if the divider is 0, then the algorithm would return an error and the certificate shows that division by 0 is not possible. A (ordinary) certifying algorithm for division might act as the strongly certifying algorithm and additionally is allowed to return anything as long as the certificate is correct. For instance, if we want to divide 0

[2]The certificate is also referred to as the *witness* in [MMNS11].

by 0 and the algorithm outputs 0 the certificate check would still be correct, as $0 = 0 \cdot 0$. A weakly certifying algorithm is additionally allowed to run without halting if we divide by 0.

For the rest of this thesis, we will only consider the case that the precondition is trivial (i.e., it is always satisfied). If the precondition is trivial, then all certifying algorithms are strongly certifying, as the exceptions for the other types can only occur when the precondition is not satisfied. In fact, in [MMNS11, Theorem 5] it is shown that any deterministic algorithm with a trivial precondition has a strongly certifying algorithm for the same problem. Hence, all the algorithms we consider from now on are strongly certifying.

Although the definitions for certifying algorithms only give guarantees about the theoretical algorithm, we can not get any guarantees about the implementation of the algorithm as software running on hardware. If we have bugs in the implementation, then we do not have any guarantees about the output or the certificate. Moreover, if there are no bugs in the implementation, it can still happen that we run into resource limits (e.g., not enough free memory) and do not produce a certificate or output at all. However, if the algorithm has correctly been implemented and enough resources are available, then the theoretical guarantees of the algorithms can be transferred.

On the one hand, this implies that if the implementation returns an output and a certificate and the checker verifies that the output is correct, then the theoretical guarantees of the algorithm transfer and the following holds. So for a strongly certifying algorithm we know that if the input did not satisfy the precondition, then the implementation correctly detected this or correctly computed an output that satisfies the postcondition.

On the other hand, if the checker rejects the output using the certificate, then there could be multiple reasons why this is the case. For instance possible reasons could be that the implementation computed an incorrect output, the implementation computed an incorrect certificate for a correct output, the implementation was prematurely terminated, or the implementation of the checker has a bug. Hence, the checker can also reject a correct output.

Even if the checker rejects, the certification process can be useful to detect the problem. A checker can be designed in a way that it not just accepts the output of rejects. It can give a reason why it rejected, which can aid in the process of debugging where the implementation of the certifying algorithm went wrong.

2.4.1 Certifying Algorithms for SAT

As the SAT problem, which was briefly introduced in Section 2.2.1, is decidable, there exist deterministic algorithms that are guaranteed to terminate for any propositional formula deciding if the formula is either satisfiable or not. We will only consider algorithms that are deterministic and are guaranteed to terminate, e.g., we will not consider local search algorithms. Since for any such algorithm there exists an equivalent strongly certifying algorithm [MMNS11, Theorem 5] with a constant factor overhead running time, the goal is to only incur a constant factor overhead for making an algorithm certifying.

A certificate for algorithms that decide the SAT problem is usually different depending on the satisfiability of the formula. If the algorithm outputs satisfiable, then the certificate is a solution that satisfies the formula, where a partial assignment that trivializes the formula is sufficient. If the algorithm outputs unsatisfiable, then the certificate is a proof showing that it is impossible to satisfy the formula. As, this direction is the interesting case, certification in the SAT community is commonly referred to as *proof logging*. This proof can take different forms and many formats have been proposed in the last couple of decades, which are discussed in detail in [Heu21]. We will briefly discuss different proof formats of unsatisfiability in chronological order.

Van Gelder [VG02] proposed to certify unsatisfiability by a *resolution proof*. A resolution proof [Bla37, DP60, DLL62, Rob65] starts with the clauses in the formula and derives new clauses using the *resolution rule* from Section 2.3. The resolution proof can be extracted from the clause learning procedure of CDCL.

Goldberg and Novikov [GN03] instead suggested a proof format based on so-called *reverse unit propagation (RUP)*. The assignment obtained at the end of unit propagation can be interpreted as the necessary assignments resulting from the starting assignment to avoid falsifying the formula. Hence, if we encounter the situation that an assignment obtained by unit propagation falsifies a clause, then the original assignment can never be extended to a satisfying assignment of the formula, which is called a *conflict*. Reverse unit propagation shows that a clause $C \doteq \ell_1 \vee \cdots \vee \ell_k$ is implied by the formula by doing unit propagation that starts with an assignment that satisfies the negated clause $\neg C \doteq \overline{\ell}_1 \wedge \cdots \wedge \overline{\ell}_k$ and has to result in a conflict. This conflict shows that the only way the formula could be satisfied is if $\ell_1 \vee \cdots \vee \ell_k \doteq C$, which shows that we can add the clause C to the formula. To denote that $F \cup \{\neg C\}$ unit propagates to conflict, we use $F \vdash_1 C$. For sets of constraint F', the notation $F \vdash_1 F'$ means that $F \vdash_1 D$ for each $D \in F'$. Using this notation we can formally state that the constraint C can be derived by reverse unit propagation from the formula F if

$$F \vdash_1 C\,. \tag{1}$$

The performance of unit propagation depends on the number of clauses to keep track of, as we do not know which clauses propagate. Hence, it became clear that deletions of clauses are crucial [HHW14]. Alternatively, this issue can be fixed by providing hints which constraints propagate in which order [WHH14, CHH^{+}17]. Systems that allow to delete constraints will be prefixed with deletion (D), e.g., reverse unit propagation (RUP) with deletion becomes deletion reverse unit propagation (DRUP). All the systems discussed in this section can be extended with a deletion rule.

While RUP can certify CDCL, dual pre- and inprocessing techniques can not be certified using RUP. The main issue is that RUP can only derive implied clauses, i.e., clauses that do not change the set of solutions to a formula. To certify such pre-

and inprocessing, Järvisalo et al. [JHB12] proposed proofs based on the *resolution asymmetric tautology (RAT)* rule, which guarantees satisfiability-equivalence of the formula and the formula with the added clause. The idea of the rule is to extend RUP[3] with one step of resolution. A clause C can be added to a formula F by RAT if there is a literal $\ell \in C$ such that resolvent of C and any clause $C' \in F$ where $\overline{\ell} \in C'$ is RUP. The correctness of this rule can be seen by considering that C is not satisfied by some solutions of F. Specifically, solutions that assign ℓ to false are no longer solutions to $F \wedge C$. Hence, we have to guarantee that all clauses in F that might be falsified by removing these solutions, namely clauses $C' \in F$ containing the literal $\overline{\ell}$, can still be satisfied by some other solution to the formula. This is the case if $C' \setminus \{\overline{\ell}\}$ is implied by F.

Alternatively, this can be formalized by saying that clause C with literal $\ell \in C$ can be derived by RAT from a formula F if

$$F \wedge \neg C \vdash_1 F\restriction_{\{\ell \mapsto 1\}} . \tag{2}$$

Here, the substitution of ℓ to true is called the witness and should be specified explicitly. As the proof obligations of the rule depend on the current formula, allowing to delete constraints from the formula can actually strengthen this proof rule [BT21]. While this rule does not preserve solutions with respect to propositional logic, it preserves solutions with respect to *overwrite logic*, which extends propositional logic with a so-called overwrite operator [RS18].

RAT can be generalized to the *propagation redundancy (PR)* rule [HKB17][4], where the witness can be an arbitrary assignment. The idea for this rule is that adding clause C could remove solution (if there are any) for F, but there should be at least one solution to the formula that can be extended from the assignment ρ. Hence, we have to show that $C\restriction_\rho$ and for all $D \in F$ that $D\restriction_\rho$ is implied by the formula F. Hence, we can think of the witness as the way to repair any potential solution that could have been removed. Formally, the clause C can be derived by the PR rule from a formula F given a (partial) assignment ρ if

$$F \wedge \neg C \vdash_1 (C \cup F)\restriction_\rho . \tag{3}$$

The PR rule can be even more generalized to the *substitution redundancy (SR)* rule [BT21] by allowing a substitution as the witness. Similar to PR rule the SR rule uses the witness to repair any solution removed by the added clause C. Formally, the clause C can be derived by the PR rule from a formula F given a substitution ω if

$$F \wedge \neg C \vdash_1 (C \cup F)\restriction_\omega . \tag{4}$$

Inspired by the SR rule, Rebola-Pardo [RP23] has extended overwrite logic to *mutation logic* such that SR proofs preserve solutions in mutation logic. Through

[3]The property of *asymmetric tautology (AT)* is equivalent to RUP.

[4]Redundancy in the SAT community means that adding a clause to or removing a clause from a formula does not change the satisfiability of the formula.

this formalization improvements to the SR rule became apparent which resulted in the so-called *weak substitution redundancy (WSR)* rule. The key observation of Rebola-Pardo is that SR rule can delete clauses that are no longer needed after the WSR rule application. Hence, these clauses can be removed from the proof obligations, but can still be used as premises for the proof obligations. Formally, the WSR rule states that the clause C can be derived from a formula F given a subformula $G \subseteq F$ and a substitution ω if

$$F \wedge \neg C \vdash_1 (C \cup G)\restriction_{\omega} . \tag{5}$$

The resulting formula is $G \cup C$.

While it is possible to express very advanced reasoning techniques using WSR with very short certificates that scale linear in the reasoning conducted by solvers, there are some limitations for this proof system, which we will discuss in Section 3.

3 Related Work

Some related work has already been mentioned in Section 2.4.1 with the certification formats for SAT like DRAT, propagation redundancy, and weak substitution redundancy. However, these systems cannot efficiently certify all state-of-the-art reasoning techniques. The best know approach to certify parity reasoning scales cubic in the size of the formula [PR16], while our approach scales linear [GN22]. Even though it is possible to deal with simple symmetry breaking using these systems, it is not known how to certify the full range of techniques in modern symmetry breaking that our approach can certify [BGMN23].

The *DSRUP* system [TD20] has been proposed for handling symmetries in SAT solvers with a focus on solvers that want to derive symmetric versions of clauses derived by RUP with respect to known symmetries of the formula. Hence, it is only possible to derive implied constraints with this system, which makes it impossible to support pre- and inprocessing techniques. Especially, the technique of symmetry breaking is not supported, as symmetry breaking constraints are not implied, as they remove symmetric solutions.

While these systems have been designed to certify SAT solvers, they are used in ad hoc methods to certify other problem by encoding a SAT formula that proves a desired property about the problem instance and using a SAT solver to certify that this property holds. For example, this approach is used to certify solvers for hardware model checking [YBH21, FYBH24] and model counting [CCS24, BNAH23]. The main issue with these certificates is that in order to trust the certificate, we also have to trust the encoding of the property that we are interested in into a SAT formula, which can be non-trivial. In most cases this is fixed by having formally verified code with a small trust base to generate the encoding. Another difference to our approach is that the certification is decoupled from the solving. Hence, it is impossible to predict the scaling of the certificate and an error in the certificate is not linked directly to reasoning in the solver.

When it comes to the certification of MaxSAT solvers, there are other approaches that have been studied before. *MaxSAT-Resolution* [HL06] is defined for the MaxSAT formulation with soft and hard clauses. This system is extended with the redundancy notion called *inclusion redundancy* [BBL24], which allows introducing a clause C to a formula F if for a witness ω satisfying C it holds that $F\upharpoonright_{\neg C} \supseteq F\upharpoonright_{\omega}$. This rule is weaker than the redundancy notions discussed in Section 2.4.1, but automatically preserves the optimal value of the problem. The downside with MaxSAT-Resolution is that certification is only known for branch and bound algorithms and preprocessing, and it is unlikely that this system is able to certify core-guided solving [BBL24]. Furthermore, MaxSAT-Resolution has no practical relevance, as no modern MaxSAT solver that implements certification based on MaxSAT-Resolution.

There is also more straightforward extension of propagation redundancy for MaxSAT called *cost propagation redundancy*, which was proposed by Ihalainen et al. [IBJ22]. This rule is similar to the redundance-based strengthening rule in VeriPB, but additionally allows adding new variables to the objective function. This behaviour can be simulated in the VeriPB system with a redundance-based strengthening step followed by an objective update. The main downside of the work by Ihalainen et al. is that they were not able to figure out how the condition on the objective function can be checked efficiently using only clausal reasoning.

Another commonly suggested idea for MaxSAT is to check that the optimal solution satisfies all clauses and to certify optimality by running a SAT solver on the clauses together with clauses encoding that only strictly better solutions are allowed. This approach has been evaluated in Paper A, which shows that this approach has unpredictable scaling behaviour and still requires certification of the clausal encoding that only solution strictly better solutions are allowed.

To certify the correctness of mixed integer linear programming (MIP), the *VIPR* system [CGS17] was developed. This certification system is focused on LP-based branch and cut MIP solvers. Hence, the certificate format is very specialized and does not really support any other solving technique. Additionally, VIPR does not have any notion of redundancy as known for the certification of SAT solver, which makes it impossible to certify advanced presolving techniques.

There has also been recent work that used the VeriPB system to provide certification to different kinds of solvers. There are certifying constraint programming solvers using VeriPB [MM23, MMN24, FSM+24, MM25] to certify reasoning with a wide range of constraint propagators. However, it is still an open problem to provide certification for all kinds of propagators used in a modern constraint programming solver and provide formally verified encodings of constraint programming problem into a pseudo-Boolean optimization problem. For optimal classical planning, Dold et al. [DHN+25] proposed a theoretical framework that uses the VeriPB system to certify the optimality of a plan. VeriPB has also been used to certify the correctness of the Pareto front for multi objective MaxSAT solvers [JBBJ25].

4 Pseudo-Boolean Certificates

This section focuses on the certification system studied in this thesis. We will first motivate the design principles guiding our certification system for combinatorial optimization. Then we will discuss our certification system in detail with a focus on the contribution of this thesis. Finally, some algorithms and data structures used in our reference implementation of a proof checker for our certification system.

4.1 Motivation

Our certification system is based on the cutting planes proof system. There are theoretical advantage of using cutting planes that are motivated by proof complexity, see Section 2.3. When comparing cutting planes to resolution, Haken [Hak85] showed exponential lower bounds in the number of steps required to refute the so-called pigeonhole principle formula, but cutting planes only requires polynomially many steps. Hence, using cutting planes can give exponentially shorter proofs for a formula than using resolution. However, it is possible to simulate extended resolution using DRAT [JHB12] and DRAT using extended resolution [KRH18]. Hence, DRAT is as strong as extended resolution as a proof system.

Our system can simulate DRAT and therefore also extended resolution, but it is not immediately clear that it could be exponentially stronger. However, as highlighted in Section 2.4, polynomial improvements in size of the certificate are important for certifying algorithms to achieve linear sized certificates. Additionally, Kołodziejczyk and Thapen [KT24] showed that the dominance-based strengthening can simulate the proof system G_1, which is above extended resolution and possibly hints towards our system being stronger than extended resolution.

Besides the theoretical advantages of cutting planes, a proof system using pseudo-Boolean constraints has the advantage that the constraints are more expressive than clauses, when comparing to SAT based certification approaches. Many problems and reasoning can be encoded more concisely. A trivial example is an at-most-one constraint stating that at most one literal of a set of n literals is true, which can be represented with one pseudo-Boolean constraint but requires n^2 many clauses. While it is possible to represent such constraints using fewer clauses, the pseudo-Boolean constraint is still more concise and easier to grasp. It could even be discussed to lift the restrictions imposed by pseudo-Boolean constraints and more complex constraints. However, there is a trade-off between the expressiveness of the constraints and the complexity of how to handle constraints inside a checker to be sure that the checker handles them correctly.

We propose also that certification should be done in a unified multipurpose system instead of a specialized system for each component of the algorithm. For instance, image Having a certificate for preprocessing in one format and the certificate for the main solver in another format. The problem with this approach is that we need guarantees between the interplay between different certificates. This could be achieved by having a checker that can deal with both formats, but than the checker internally has to switch between different representation, e.g., if one format reasons on a graph and the other uses pseudo-Boolean constraints. This has the downside that it increases the complexity of the checker, which makes it more difficult to trust the checker. Alternatively, an interface between proofs could be defined, so that the different proofs can be checked by different checkers, which keeps each checker simple. We are actually pioneering this approach with the output section, where we can certify guarantees on a reformulated problem that can be used as input to the next checker.

Another advantage of having one unified multipurpose system is that solver authors can trust that the certification system is strong enough to certify new reasoning techniques added to the solver. Moreover, new tool, like preprocessors or symmetry breaking tools, can just be added to the solver without hassle if the tool uses the same system as the solver.

Additionally, our philosophy is that the certification should follow the reasoning of the solver as close as possible, which is contrast to just certifying the result by possibly using an independent approach. There are several advantages to this that we will discuss in the rest of this section. If the certification follows the reasoning closely, then we can get upper bounds on the size of the certificate and the additional time required to write the certificate. This is required to get efficient certifying algorithms in the sense of McConnell et al. [MMNS11].

The fact that the certificate is written while the solver is running also enables us to have certification for anytime solvers that can be interrupted and stopped arbitrarily. For instance, a solver that generates the certificate after solving would not generate any certificate if it were stopped suddenly. However, if a solver that is generating a certificate while solving is stopped, then it can finish up the certificate in the same routine that is printing the anytime result.

The closer the certificate follows the reasoning in the solver and the more detail the certificate contains, the better it can be used for detecting bugs in the reasoning of the solver. This approach can even detect bugs on instances where the solver still returns the correct result but the reasoning that led to this result is erroneous. Detecting bugs even if the returned result is correct and without even knowing what the correct result should be makes software testing and the approach of fuzzing [ZWCX22] for software testing extremely powerful. If a bug occurs, then a detailed certificate can even help to find the cause of the issue by tracing back the certificate.

A detailed certificate can also be used to deeper understand the reasoning performed by the solver. In our approach it is even possible to annotate the certificate with comments to see which parts of the certificate came from which part of the solver. This approach could even be used to extract why the solver came to this conclusion in a concise human-readable form, so that users can understand the reasons for the returned result.

All of this should make it clear that there are advantages of having a unified multipurpose certification systems that can closely follow the reasoning of the solver. Even though we have seen advantages of using pseudo-Boolean constraints to represent the reasoning, the specific format of the constraints can be debated and more general constraints than pseudo-Boolean constraints might be advantageous.

4.2 Our Pseudo-Boolean Proof System

The pseudo-Boolean proof system VeriPB was pioneered by Gocht et al. in the course of several publications [EGMN20, GMN20, GMM⁺20, GN22, BGMN23, Goc22]. We will first discuss some general design principles of the theoretical proof system and the representation of the proof rules in file format. Then we will review some prior work on the proof system before we will discuss the contributions to the certification system made by this thesis in detail. In this section we only focus that the system is correct and in Section 4.3 we will discuss efficient algorithms for checking the correctness of rule applications.

Our system is based on the cutting planes proof system, which means that the atoms of reasoning are pseudo-Boolean constraints. In the proof system we always assume the constraints are stored and treated in normalized form while the format allows stating non-normalized constraints, these are normalized directly after parsing. Each constraint in the proof is assigned a consecutive ID, starting with the constraints in the original problem and continuing with the constraints introduced by the proof.

The constraints known at any point in the proof are partitioned into a *core set* and a *derived set* of constraints. The core set is initialized to the constraints of the input problem. The idea of the core set is that we can have guarantees on how these constraints relate to the input constraints, e.g., they are equisatisfiable. The derived set is initialized to be empty. All constraints added by the proof rules are added to the derived set. We do not have to guarantee anything for the constraints in the derived set, except that they are derived by a valid rule application. Järvisalo et al. [JHB12] referred to the core set as the irredundant set and to the derived set as the redundant set.

The completeness of our proof system follows from the completeness of the cutting planes proof system and that any standard cutting planes proof is also a proof in our system. To show the soundness of our proof system, especially for the new rules, we use the same notations and definitions used in by Bogaerts et al. [BGMN23], but extend it slightly to show the correctness of the new rules. The proof system is a sequence of *proof configurations* $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, s)$, where

- $\mathcal{C}$ is the core set of constraint
- $\mathcal{D}$ is the derived set of constraints
- f^* is the current objective function
- $\mathcal{O}_\succeq$ is the currently active order
- $\vec{z}$ is the vector of literals the active order is initialized over
- g is a Boolean variable indicating stronger guarantees on the core set
- u is the best objective value recorded

- v is the best objective value recorded while $g = \top$ (*incumbent value*)
- s is a Boolean variable indicating if the strengthening-to-core mode is active.

The configuration closely resembles the global state maintained by the checker. The initial proof configuration for an optimization problem with objective f and set of constraints F is $(F, \emptyset, f, \emptyset, \emptyset, \top, \infty, \infty, \bot)$.

Definition 1 (cf. [BGMN23, Definition 1])**.** For an optimization problem with objective f and set of constraints F the configuration is $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is (F, f)*-valid* if it holds that

1. For every $u' < u$, it holds that if $F \cup \{f \leq u'\}$ is satisfiable, then $\mathcal{C} \cup \{f^* \leq u'\}$ is satisfiable.
2. For every total assignment ρ satisfying $\mathcal{C}$, there exists a total assignment ρ' satisfying $\mathcal{C} \cup \mathcal{D} \cup \{f^*(\rho) \geq f^*(\rho')\} \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_{\rho}, \vec{z}\restriction_{\rho'})$.
3. If $v < \infty$, then $F \cup \{f \leq v\}$ is satisfiable.
4. For every $v' < v$, it holds that if $\mathcal{C} \cup \{f^* \leq v'\}$ is satisfiable, then $F \cup \{f \leq v'\}$ is satisfiable.
5. If $s = \top$, then any total assignment satisfying $\mathcal{C}$ also satisfies $\mathcal{C} \cup \mathcal{D}$.

All rules in our proof system preserve (F, f)-validity, which we will show later in the thesis for the rules, where this does not follow from directly. The following theorem combines two theorems by Bogaerts et al. and establishes the relationship between (F, f)-validity and the soundness of the proof system. The theorem has been adjusted slightly to incorporate the changes to the proof system in this thesis compared to the one used by Bogaerts et al.

Theorem 1 (cf. [BGMN23, Theorem 2 and 3])**.** *Let F be a set of pseudo-Boolean constraints and f be a pseudo-Boolean objective. If $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is an (F, f)-valid configuration, then the following holds:*

i) If $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$ and $u = \infty$, then F is unsatisfiable.

ii) If F is unsatisfiable, then $v = \infty$.

iii) Let $lb \leq u$. If $\mathcal{C} \cup \mathcal{D}$ contains the constraint $f^* \geq lb$, then any solution ρ satisfying F has objective value $f(\rho) \geq lb$.

iv) If F is satisfiable, then there is a solution ρ satisfying F with objective value $f(\rho) \leq v$.

Proof. We prove the correctness of the theorem item by item. For Item i, assume for contradiction that F is satisfiable. By Item 1 in Definition 1, the constraints in $\mathcal{C}$ are satisfiable, as $u' < \infty$ can be chosen large enough. Finally, by Item 2 in Definition 1, $\mathcal{C} \cup \mathcal{D}$ is also satisfiable, which contradicts that $\mathcal{C} \cup \mathcal{D}$ contains the constraint $0 \geq 1$.

For Item ii, assume for contradiction that $v < \infty$. By Item 3 in Definition 1, $F \cup \{f \leq v\}$ is satisfiable. Hence, F itself is also satisfiable, which is a contradiction to F being unsatisfiable.

For Item iii, assume for contradiction that there is a solution ρ' satisfying F with $f(\rho') < lb$, hence ρ' satisfies $F \cup \{f \leq lb - 1\}$. By $lb \leq u$ and Item 1 in Definition 1, we have that $\mathcal{C} \cup \{f^* \leq lb - 1\}$ is satisfiable. Let $\hat{\rho}$ be a solution satisfying $\mathcal{C} \cup \{f^* \leq lb - 1\}$, then $f^*(\hat{\rho}) < lb$ and $\hat{\rho}$ also satisfies $\mathcal{C}$. By Item 2 in Definition 1, there exists a solution $\tilde{\rho}$ that satisfies $\mathcal{C} \cup \mathcal{D}$ and has an objective value $f^*(\tilde{\rho}) < lb$, which is a contradiction to $\mathcal{C} \cup \mathcal{D}$ containing the constraint $f^* \geq lb$, as $f^* \geq lb$ is falsified by $\tilde{\rho}$.

For Item iv, we consider two cases. If $v = \infty$, then there is a solution satisfying F, since F is satisfiable, and any solution ρ' has objective value $obj(\rho') \geq \infty$. If $v < \infty$, then $F \cup \{f \leq v\}$ is satisfiable by Item 3 in Definition 1. This immediately gives us that any solution ρ' satisfying $F \cup \{f \leq v\}$ satisfies F and has an objective value $f(\rho') \leq v$. $\square$

Items 4 and 5 in Definition 1 is not required to prove Theorem 1, but is required later to show that our proof rules preserve (F, f)-validity. The Items i and ii are mainly relevant for decision instances, while Items iii and iv are only relevant for optimization instances.

4.2.1 Rules from Previous Work

In this section we will discuss the rules that have remained unchanged compared to Bogaerts et al. [BGMN23] and Gocht [Goc22]. If rules have been changed since the work by Bogaerts et al., we will detail the updated rules in Section 4.2.2 and omit it in this section.

Implicational Rules The first set of rules are the rules from the *cutting planes* proof system and rules that are just syntactic sugar for cutting planes derivations. Our proof system supports *reverse unit propagations (RUP)* similar to RUP in Section 2.4 but using pseudo-Boolean unit propagation. Another rule is *syntactic implication*, which check if a constraint C can be derived from another constraint D by adding literal axioms, saturating with respect to the degree of D, and adding more literal axioms. All of these rules have the effect that the configuration changes from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ trivially preserve (F, f)-validity, see [BGMN23, Section 3.1].

Sanity Check Rules There are also some rules for sanity checks that do not modify the configuration, hence they are trivially sound. Their purpose is to check if the configuration is as expected, so that if a discrepancy occurs between the proof and the solver, the check immediately detects this. For a specific constraint C and the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$, we can check if a *syntactically equivalent* or *syntactically implied* constraint to C is in $\mathcal{C} \cup \mathcal{D}$. If this is not the case, the proof will fail.

Move to Core A constraint C can be moved from the derived set to the core set, but not vice versa. This changes the configuration from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, s)$ to $(\mathcal{C} \cup \{C\}, \mathcal{D} \setminus \{C\}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, s)$. This preserves (F, f)-validity, since the core set gets more constrained, which preserves the required guarantees.

Order Change We can *change the order* by specifying a possibly empty pseudo-Boolean formula $\mathcal{O}'_\succeq(\vec{u}, \vec{v})$ and a list of literals $\vec{z}'$ that should be compared, where $\vec{u}$, $\vec{v}$, and $\vec{z}$ have the same size. The formula $\mathcal{O}'_\succeq(\vec{u}, \vec{v})$ has to be shown to be reflexive, i.e., $\emptyset \vdash \mathcal{O}'_\succeq(\vec{u}, \vec{u})$, and transitive, i.e., $\mathcal{O}'_\succeq(\vec{u}, \vec{v}) \cup \mathcal{O}'_\succeq(\vec{v}, \vec{w}) \vdash \mathcal{O}'_\succeq(\vec{u}, \vec{w})$, by only using implicational rules, where $\vec{w}$ has the same size as $\vec{v}$. For this rule to be sound we require that the derived set $\mathcal{D}$ is empty. Hence, we can change the configuration from $(\mathcal{C}, \emptyset, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, s)$ to $(\mathcal{C}, \emptyset, f^*, \mathcal{O}'_\succeq, \vec{z}', g, u, v, s)$. This rule is sound, since all items except Item 2 in Definition 1 are trivial and Item 2 holds because the derived set is empty and any satisfying assignment to $\mathcal{C}$ is also a satisfying assignment to $\mathcal{C} \cup \mathcal{D}$.

Solution Logging The *solution logging* rule (or objective bound update rule in [BGMN23]) can be used to update the best recorded objective values and to derive a strict upper bounding constraint on the objective function. Given a total assignment ρ that satisfies $\mathcal{C}$, we can change the configuration from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, s)$ to $(\mathcal{C} \cup \{f^* \leq f^*(\rho) - 1\}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, f^*(\rho), f^*(\rho), s)$ if $g = \top$ and to $(\mathcal{C} \cup \{f^* \leq f^*(\rho) - 1\}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, f^*(\rho), v, s)$ if $g = \bot$. Thus, the incumbent value v is only updated if $g = \top$. If ρ is only a partial assignment, then ρ is propagated with respect to $\mathcal{C} \cup \mathcal{D}$, and if the assignment is then still not total, then the proof is declared incorrect. As shown by [BGMN23], this rule preserves (F, f)-validity, since Items 1, 2, 4, and 5 are trivial and Item 3 follows from Item 4.

Redundance-Based Strengthening We are now discussion two rules to add constraints that are not implied. These rules behave differently when the strengthening-to-core mode $s = \top$, which is explained in detail in Section 4.2.2. First, *redundance-based strengthening* [GN21] is a generalization of substitution redundancy [BT21] to pseudo-Boolean reasoning allowing arbitrary implicational proofs instead of just unit propagation. For a constraint C, redundance-based strengthening changes the configuration from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, \bot)$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, f^*, \mathcal{O}_\succeq, \vec{z}, g, u, v, \bot)$ given a witness substitution ω and cutting planes proofs showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_\succeq(\vec{z}, \vec{z}\restriction_\omega) \tag{6}$$

using only implicational rules. A proof that redundance-based strengthening preserves (F, f)-validity can be found in [BGMN23, Proposition 7].

Dominance-Based Strengthening Second, the *dominance-based strengthening* rule [BGMN23] generalizes redundance-based strengthening even more and makes use of applying the witness iteratively to show its correctness. For a constraint CC, dominance-based strengthening changes the configuration from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \bot)$ to $(\mathcal{C}, \mathcal{D} \cup \{C\}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \bot)$ given a witness substitution ω and cutting planes proofs showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash \mathcal{C}\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_\omega) \tag{7}$$

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \mathcal{O}_{\succeq}(\vec{z}\restriction_\omega, \vec{z}) \vdash \bot \tag{8}$$

using only implicational rules. Instead of (8) giving a cutting planes proof for

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \cup \{f^*\restriction_\omega \geq f^*\} \vdash \bot \tag{9}$$

is also sufficient for a valid dominance-based strengthening step. A proof showing that dominance-based strengthening preserves (F, f)-validity can be found in [BGMN23, Proposition 14].

Deletion The rules discussed so far only allow us to add constraints, but for the performance of the checker and the strength of the proof system, the system also supports the *deletion* of constraints. If we delete a constraint C from the derived set $\mathcal{D}$, then the configuration changes from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ to $(\mathcal{C}, \mathcal{D} \setminus \{C\}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ without any checks, as we do not guarantee anything for $\mathcal{D}$. If we delete a constraint C from the core set $\mathcal{C}$, then we can either use checked or unchecked deletion to remove C from $\mathcal{C}$. For *checked deletion*, we have to give a substitution witness ω and cutting planes proofs showing

$$(\mathcal{C} \setminus \{C\}) \cup \{\neg C\} \vdash \mathcal{C}\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_\omega) \tag{10}$$

using only implicational rules, i.e., C can be derived by redundance-based strengthening for $\mathcal{C} \setminus \{C\}$. Checked deletion changes the configuration from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ to $(\mathcal{C} \setminus \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$. For *unchecked deletion*, we only need to check that if the $\mathcal{O}_{\succeq} \neq \emptyset$, then $\mathcal{D} = \emptyset$, but we lose the stronger guarantees g, setting $g = \bot$, and we need that strengthening-to-core $s = \bot$. Thus, in terms of configurations, we transition from $(\mathcal{C}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \bot)$ or $(\mathcal{C}, \mathcal{D}, f^*, \emptyset, \emptyset, g, u, v, \bot)$ to the configuration $(\mathcal{C} \setminus \{C\}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, \bot, u, v, \bot)$ or $(\mathcal{C} \setminus \{C\}, \mathcal{D}, f^*, \emptyset, \emptyset, \bot, u, v, \bot)$, respectively. More details about deletion and propositions showing that deletion preserves (F, f)-validity can be found in [BGMN23, Section 3.4].

Convenience Rules To make it easier to keep track of deletions in branch and bound algorithms, constraints can be assigned a *level*, which is a mapping from constraints to non-negative integers. When wiping a level *lvl*, all constraints with level *lvl* or higher are deleted, which is just syntactic sugar for deleting all these constraints after each other.

There is an additional rule for debugging which allows adding arbitrary constraints to the derived set. However, this rule does not preserve the (F, f)-validity and the proof checker will warn the user if this rule is used that the proof is invalid.

4.2.2 Extensions to the System by This Thesis

In the included papers the proof system has been extended with the following rules. Some rules are also just extensions of already existing rules.

Objective Equivalence There is an additional rule for checking that the current objective f^* in the configuration $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is *syntactically equivalent* to the objective specified in the rule. This rule does not change the configuration and trivially preserves (F, f)-validity.

Objective Update The main contribution of Paper IV to the proof system is the *objective update* rule that allows to change the objective. Using the objective update we can transition from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ to $(\mathcal{C}, \mathcal{D}, f', \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ given cutting planes proofs showing that

$$\mathcal{C} \vdash \{f^* \geq f'\} \cup \{f' \geq f^*\} \tag{11}$$

using only implicational rules. This shows that $f^* = f'$. While the objective update is still sound if $f^* \geq f'$ is derived from $\mathcal{C} \cup \mathcal{D}$, we require for simplicity that it derived by $\mathcal{C}$ only. To keep the size of the proof as small as possible, there are two ways to specify the new objective. The first way is to directly specify the updated objective f'. The second way is to specify the difference between the updated objective f' and the current objective f^*, i.e., the linear form $f' - f^*$. Proposition 2 argues that the objective update preserves (F, f)-validity.

Proposition 2. *If $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is (F, f)-valid, and we use the objective update rule, then $(\mathcal{C}, \mathcal{D}, f', \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is also (F, f)-valid after applying the rule.*

Proof. Items 3 and 5 in Definition 1 is trivially preserved, as it not affected by the objective update. Item 1 in Definition 1 is satisfied, as $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is (F, f)-valid and $f^* = f'$. Thus, since $\mathcal{C} \cup \{f^* \leq u'\}$ is satisfiable, $\mathcal{C} \cup \{f' \leq u'\}$ is also satisfiable. Similarly, Item 4 in Definition 1 is also satisfied, since $\mathcal{C} \cup \{f^* \leq v'\}$ is satisfiable exactly when $\mathcal{C} \cup \{f' \leq v'\}$ is satisfiable. For any total assignment ρ, $f^* = f'$ guarantees that $f^*(\rho) = f'(\rho)$. Hence, Item 2 in Definition 1 is satisfied, since $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ is (F, f)-valid and $f'(\rho) = f^*(\rho) \geq f^*(\rho') = f'(\rho')$. □

Proof Output We have introduced support in the certification system to state the results certified by the proof in a *proof footer*. This improves on the ad-hoc certification of optimizations problems in [BGMN23] by checking if the optimal value obtained by the solver matches the optimal value certified by the proof.

The first part of the footer states the *output* of the proof and its guarantees. Let $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ be the configuration at the end of the proof. The output can be given externally, e.g., in form of an additional file, or is implicitly defined as the core set $\mathcal{C}$. If the output is given externally with the objective f' and constraints F', then we check that $f' \doteq f^*$ for each constraint $C' \in F'$ that is a constraint $C \in \mathcal{C}$ such that $C' \doteq C$ and vice versa. The most trivial guarantee is *derivable* and just states that the output formula with objective f' and constraints F' can be derived from the original formula with objective f and constraints F using the proof system preserving (F, f)-validity. The stronger guarantees are *equisatisfiable* and *equioptimal*, which are used for decision and optimization problems, respectively. The guarantee equisatisfiable guarantees that the output constraints F' are satisfiable if and only if the original constraints F are satisfiable. The guarantee equioptimal guarantees that the output problem with objective f' and constraint F' has the same optimal value as the input problem with objective f and constraint F. The guarantees equisatisfiable and equioptimal can only be used if $s = \top$. This makes it possible to have a standalone certificate for problem reformulations. It is also possible to specify that the proof has no output.

Proof Conclusion The second part of the footer is the *conclusion* of the proof. Four types of conclusion are supported, which also includes that there is no conclusion. Let $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, s)$ be the configuration at the end of the proof. If we are just interested in deciding the satisfiability a problem with constraints F, then the conclusion can either be satisfiable or unsatisfiable. If we conclude with *unsatisfiable*, then it is checked if $\perp \in \mathcal{C} \cup \mathcal{D}$. To use the conclusion *satisfiable*, it must hold that either $v < \infty$ or that a solution specified together with the conclusion satisfies F. For optimization instances we are able to conclude with *bounds* on the optimal value of the objective function f, which can also be specified to be ∞. For the lower bound *lb*, it is checked that the lower bounding constraint $f \geq lb \in \mathcal{C} \cup \mathcal{D}$. For the upper bound *ub*, it is checked whether $v \leq ub$ or that a solution ρ specified together with the conclusion satisfies F and $f(\rho) \leq ub$.

Hinted Reverse Unit Propagation Similar to the LRAT format [CHH+17], reverse unit propagation now supports hints pointing towards the constraints that have to be propagated to derive the contradiction. This can significantly speed up the checker, as it only has to look at a subset of the database to detect the propagations. Additionally, it is also supported that only some RUP steps are annotated with hints and the other RUP steps do not have any hints to make the format more flexible, which is similar to the FRAT format [BCH21].

Strengthening-to-Core mode Finally, in Paper III the *strengthening-to-core mode* is introduced. This mode is disabled by default and can be turned on and off in the proof. The strengthening-to-core mode can be activated if the derived set $\mathcal{D}$ is empty, hence it can change the configuration from $(\mathcal{C}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \perp)$ to $(\mathcal{C}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$. The strengthening-to-core mode can be disabled at any time transitioning from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ to $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \perp)$. The idea of this mode is that constraints derived by strengthening rules are added immediately to the core set, which guarantees that any solution satisfying $\mathcal{C}$ also satisfies $\mathcal{D}$. Enabling strengthening-to-core preserves (F, f)-validity, since Items 1 to 4 in Definition 1 are not affected. Item 5 in Definition 1 is preserved, since $\mathcal{D} = \emptyset$, so that any satisfying assignment to $\mathcal{C}$ also satisfies $\mathcal{C} \cup \mathcal{D}$.

To maintain that the proof system is sound while the strengthening-to-core is enabled, we need to be careful when deleting constraints from the core set $\mathcal{C}$. When using checked deletion, then the substitution witness ω used for (10) has to be trivial, which means validity of rule requires an implicational cutting planes proof showing that

$$(\mathcal{C} \setminus \{C\}) \cup \{\neg C\} \vdash C \,. \tag{12}$$

When using unchecked deletions from the core set, then the derived set $\mathcal{D}$ must be empty. This means we transition from $(\mathcal{C}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ to $(\mathcal{C} \setminus \{C\}, \emptyset, f^*, \mathcal{O}_{\succeq}, \vec{z}, \bot, u, v, \bot)$.

The reasons for this restriction of deletion is that otherwise it would be possible to derive contradiction from any satisfiable formula F. To see that this is possible, we consider that the strengthening-to-core mode is enabled. For a variable y that is not used in F, we derive $y \geq 1$ using redundance-based strengthening with the witness $\{y \mapsto 1\}$, which is added to $\mathcal{C}$. Using a cutting planes derivation we can derive $y \geq 1$, which is added to the derived set. Without the restrictions, we can delete $y \geq 1$ from $\mathcal{C}$ by either checked deletion using the witness $\{y \mapsto 1\}$ or unchecked deletion. Then we derive $\overline{y} \geq 1$ by redundance-based strengthening using the witness $\{y \mapsto 0\}$. Finally, we derive the contradiction $0 \geq 1$ using cutting planes by adding $y \geq 1$, which is still in the derived set, and $\overline{y} \geq 1$.

If strengthening-to-core is enabled, then the redundance-based strengthening rule deriving constraint C transitions from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ to $(\mathcal{C} \cup \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$. This requires that we are given a substitution witness ω and cutting planes proofs showing that

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_\omega) \tag{13}$$

using only implicational rules. The advantage of (13) compared to (6) is that we no longer have to derive $\mathcal{D}\restriction_\omega$. Proposition 3 shows that this updated rule still preserves (F, f)-validity.

Proposition 3. *If $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ is (F, f)-valid, we can use the redundance-based strengthening rule to transition to $(\mathcal{C} \cup \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$, which is also (F, f)-valid.*

Proof. We assume that $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ is (F, f)-valid. For Item 1 in Definition 1, we can modify the proof by Bogaerts et al. [BGMN23, Proposition 7]. We know that for every $u' < u$, that if $F \cup \{f \leq u'\}$ is satisfiable, then $\mathcal{C} \cup \{f^* \leq u'\}$ is also satisfiable. We now show that $\mathcal{C} \cup \{f^* \leq u'\} \cup \{C\}$ is also satisfiable by constructing assignments ρ' satisfying $\mathcal{C} \cup \{f^* \leq u'\} \cup \{C\}$. Let ρ be a total assignment satisfying $\mathcal{C} \cup \{f^* \leq u'\}$. If ρ also satisfies $\mathcal{C} \cup \{f^* \leq u'\} \cup \{C\}$, then we use $\rho' = \rho$.

Otherwise, we know that ρ satisfies $\neg C$. Since ρ satisfies $\mathcal{C}$, ρ also satisfies $\mathcal{C} \cup \mathcal{D}$ by Item 5 in Definition 1. Therefore, ρ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$. By (13), ρ also satisfies $(\mathcal{C} \cup \{C\})\restriction_\omega \cup \{f^* \geq f^*\restriction_\omega\} \cup \mathcal{O}_{\succeq}(\vec{z}, \vec{z}\restriction_\omega)$. Using $\rho' = \rho \circ \omega$, it holds that

$((\mathcal{C} \cup \{C\})\restriction_{\omega})\restriction_{\rho} = (\mathcal{C} \cup \{C\})\restriction_{\rho\circ\omega}$, i.e., ρ' satisfies $\mathcal{C} \cup \{C\}$. Finally, ρ' also satisfies $f^* \geq u'$, since ρ satisfies $f^* \geq f^*\restriction_{\omega}$ and $f^*\restriction_{\rho'} = f^*\restriction_{\rho\circ\omega} \leq f^*\restriction_{\rho} \leq u'$.

Items 2, 4, and 5 in Definition 1 is trivially preserved, as any assignment satisfying $\mathcal{C} \cup \{C\}$ also satisfies $\mathcal{C}$. Item 3 in Definition 1 is preserved, as it does not depend on $\mathcal{C}$. □

For dominance-based strengthening with strengthening-to-core, we transition from $(\mathcal{C}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$ to $(\mathcal{C} \cup \{C\}, \mathcal{D}, f^*, \mathcal{O}_{\succeq}, \vec{z}, g, u, v, \top)$. This requires that we are given a substitution witness ω and proofs showing that (7) and either (8) or (9) hold. Since the derived constraint is added to the core set instead of the derived set, the proof showing that dominance-based strengthening preserves (F, f)-validity also shows that the rule preserves (F, f)-validity if strengthening-to-core is enabled.

Kernel Proof Format Finally, to make efficient formally verified proof checkers feasible for Papers V and VI, we have defined two variants of the proof format. The *augmented format* contains all the rules, whereas the *kernel format* only contains a subset of the rules in the VeriPB format. In the kernel format all RUP steps must be annotated with hints, syntactic implication is not allowed, all solutions logged must be total assignments, and constraints required for the conclusion must be referenced explicitly.

4.3 Pseudo-Boolean Proof Checking Tool

For the discussion of implementation details, we will focus on the algorithms and data structures used in the reference implementation VeriPB, which also supports the elaboration of proofs.[5] The other major implementation of a checker for the VeriPB system is the formally verified proof checker CakePB.[6]

The proof checker VeriPB uses many specialized data structures to improve the running time for checking proof, e.g., specialized data structures to do unit propagation for different kinds of constraints [Dev20b].

The main contribution of Paper V is the elaboration of an augmented proof to a kernel proof that the formally verified proof checker CakePB can handle. This can basically be thought of as certification for the reasoning performed by VeriPB in a format that CakePB can check.

[5]The source code of VeriPB is available at `https://gitlab.com/MIAOresearch/software/VeriPB`.

[6]The source code and correctness proofs of CakePB are available at `https://github.com/CakeML/cakeml/tree/master/examples/pseudo_bool`, and precompiled binaries of CakePB are available at `https://gitlab.com/MIAOresearch/software/cakepb`.

5 Main Results of This Thesis

In this section an overview of the papers included in this thesis is given to highlight the contribution of this thesis to the field. The contributions of each paper are detailed, but some papers have overlapping topics.

There are several ways to group the included papers together. The Papers I to IV do not use formal verification, hence the results obtained from the verification process do not have any formal guarantee. Papers V and VI use the formally verified proof checker CakePB. Papers II, III, and VI are about solving MaxSAT. Papers I and IV are about solving pseudo-Boolean problems. Papers IV and VI are both about presolving and preprocessing techniques that are used in combinatorial optimization.

5.1 Summary of Paper I

Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version 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.

Some MaxSAT and all pseudo-Boolean solvers that are based on SAT solvers encode pseudo-Boolean constraint into clauses [ES06, MML14, SN15, PRB18]. Additionally, for modelling it can be easier to state a pseudo-Boolean optimization problem and then use a tool to encode the pseudo-Boolean constraints into CNF [PS15]. While certification for the SAT solvers is well-established [Heu21], it has remained out of reach to certify encoding pseudo-Boolean constraints into CNF.

In this paper, we show how algorithms for encoding pseudo-Boolean constraints into CNF can be made certifying. We provide a general framework that can be used to certify the correctness for different encodings. The framework provides a skeleton algorithm, which only requires filling in the details for each component specific to the encoding. To illustrate how this framework can be used, we provide certifying algorithms for sequential counter [Sin05], binary adder network [ES06], totalizer [BB03], and generalized totalizer [JMM15] encodings.

By concatenating the certificate for the correctness of the encoding and a DRAT certificate [WHH14] from a SAT solver, which is syntactically modified to be compatible with VeriPB, we can get a certificate showing that the original pseudo-Boolean constraints are unsatisfiable. The certificate for satisfiability is a solution satisfying all pseudo-Boolean constraints.

We further demonstrate how certifying PB to CNF encodings can be used to certify the correctness of the optimal value f^* obtained by MaxSAT solvers. This is done checking that the optimal solution x^* provided by the MaxSAT solver satisfies all clauses F and checking that the reported optimal value f^* matches the objective value of the optimal solution. Then we encode the pseudo-Boolean constraint $f < f^*$ that the objective function f should be strictly smaller than the optimal value f^* into clause F^*. If the SAT solver returns unsatisfiable on the formula $F \cup F^*$, then we get a certificate showing that there is no feasible solution with a better objective value than the optimal value. However, this certificate gives no guarantee that the reasoning in the MaxSAT solver is correct for this instance.

We implemented the certification inside an encodings library and changed the SAT solver Kissat [BFF+24] to output proofs in the VeriPB format. Our experimental evaluation on the benchmark instances of the pseudo-Boolean competition 2016 [Pse16] showed that our approach can be used in practice, even though there is still room for improvements. We also evaluated the certification of optimal values returned by MaxSAT solvers and verified that the optimal values obtained in the MaxSAT Evaluation 2022 [Max22] are correct.

5.2 Summary of Paper II

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.

Core-guided MaxSAT solving is one of the state-of-the-art approaches to solve the MaxSAT problem. In this paper for the first time we show how to add certification to core-guided MaxSAT solvers with all advanced techniques used in these solvers.

The certification of the calls to the SAT solver can be used out of the box by syntactically changing the output to the VeriPB format. Since the core clause returned by the SAT solver is learnt by conflict analysis, it can be derived by a RUP step. We show that the objective reformulation in the OLL algorithm can be certified using the VeriPB system. This can easily be done, as new variables introduced by reification can be represented as two pseudo-Boolean constraints per variable. For the encoding of the reification constraints into CNF, we build on our prior work in Paper I and in [VDB22]. To transfer a lower bound lb on the reformulated objective $f_{ref} \geq lb$ to the original objective $f_{orig} \geq lb$, we maintain the pseudo-Boolean constraint $f_{orig} \geq f_{ref}$. As other core-guided algorithms are very similar to the OLL algorithm, it should be straightforward to adapt our approach to any core-guided MaxSAT algorithm.

We also provide certification for improvements to the standard OLL algorithm. Specifically, we explain how core exhaustion [ABGL12], core minimization [Mar10], hardening [ABGL12], intrinsic at-most-one constraints [IMM19], lazy variable encodings [MJML14], stratification [ABGL12, MAGL11], structure sharing [IBJ21], upper bound estimation [IMM19], and weight-aware core extraction [BJ17] can be made certifying.

We implemented our certification approach into the state-of-the-art core guided MaxSAT solver CGSS [IBJ21] that uses all the aforementioned techniques. We experimentally evaluated our approach on the benchmark instances of the MaxSAT Evaluation 2022 [Max22]. During these experiments, we detected a bug in the reasoning of CGSS, which it inherited from its predecessor RC2 [IMM19]. This bug would not have been discovered by just checking the optimal solution returned by the solver. This shows that our certification approach can be successfully used to detect bugs in established tools.

After fixing this bug, we were able to confirm that our approach is usable in practice. The overhead for generating the certificate while solving the instance is very low, except for some outliers due to interfacing between Python and C++. The performance for checking the certificate was sufficient, but could be improved by further engineering the VeriPB proof checker to be more efficient.

5.3 Summary of Paper III

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "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.

Vandesande et al. [VDB22] showed how to certify the correctness of solution-improving search (SIS) MaxSAT solvers with VeriPB. However, they implemented certification for the solver QMaxSAT [KZFH12], which is no longer state-of-the-art. For instance, the modern SIS MaxSAT solver Pacose [PRB18] uses the structure of the dynamic generalized polynomial watchdog (DGPW) encoding to perform advanced without loss of generality reasoning.

While it is possible to certify the correctness of the encoding using the framework in Paper I or the work by Vandesande et al., the advanced reasoning performed using the encoding remains out of reach. To solve this issue, we introduce the idea to construct a shadow circuit over a new set of variables that mimics the circuit that is used for the DGPW encoding. Then any without loss of generality reasoning performed by the solver is certified using a shadow circuit that has the same structure except for the variables that are without loss of generality assumed to some value. Reasoning performed on the shadow circuit is transferred back to the original circuit by redundance-based strengthening mapping each original variable to its shadow variable and the assumed variables to their assumed value.

This idea requires that the variables introduced by the original encoding only appear in encoding constraints, as this trivializes all redundance-based strengthening proofgoals from core and derived set. However, the SAT solver might learn new clauses over these variables that are important for the SAT solver reasoning and have no shadow circuit counterpart. To mitigate this issue, we introduce the strengthening-to-core mode. With this mode enabled, all proofgoals from the clauses learnt by the SAT solver can be ignored, as they end up in the derived set of constraints. The strengthening-to-core mode basically enables us to use the shadow circuit approach in a complex system with many components.

Modern SIS MaxSAT solvers also employ a wide range of additional techniques to improve performance. In this paper, we also provide certifying algorithms for all techniques used in the solver Pacose. Specifically, these techniques are adder caching [BBR09, PRB18], cone-of-influence encoding [PRB18], generalized boolean multilevel optimization (GBMO) [ALM09, PRB21], and TrimMaxSAT [PRB21].

We also discuss in detail why the alternative certification approach of running the MaxSAT solver without certification, checking the solution, and creating a certificate using a SAT solver is not feasible in practice. The main reasons are:

(i) the encoding of the solution-improving constraint still need to be certified;

(ii) the running time of SAT solver and certificate size are unpredictable; and

(iii) anytime solving cannot be certified this way.

We implemented our certification approach into the MaxSAT solver Pacose For GBMO, two different approaches are used, but during preliminary experiments we noticed that only one approach is used in practice. We only implemented certification for the one approach that is actually used, but explain certification for both in the paper. The correctness and performance of our approach is experimentally evaluated on the MaxSAT Evaluation 2023 [Max23] benchmarks. The experiments show that our approach is correct, but the performance for generating the certificate is a bit slower than what would be desirable for practical usage. We identified that some overhead in generating the certificate was due to the shadow circuits. However, some overhead seems to be due to inefficient data structures, which could be improved by further engineering effort.

5.4 Summary of Paper IV

Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. "Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs". In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

Presolving is an important technique for the performance of MIP solving [ABG$^+$20]. However, the VIPR certification system [CGS17] developed to certify MIP solving is not able to certify advanced presolving techniques. The main issue with VIPR is that it cannot certify reasoning that removes feasible solutions as long as at least one optimal solution is preserved.

Since 0–1 ILP is a specialization of MIP, our idea is that we can pioneer how presolving can be certified for 0–1 ILP to pave the way for certification of presolving for MIP in the future. This allows us to use the VeriPB system to certify the correctness of the presolving techniques, as VeriPB can reason with 0–1 ILPs and has the redundance-based strengthening to deal with techniques that remove feasible solutions. In this paper, we present certification for all presolving techniques applied to 0–1 ILPs by the state-of-the-art presolver PaPILO [GGH22], which are most of the techniques implemented in PaPILO.

To enable certification for all the technique, we extend the VeriPB system with the objective update rule. This rule is required to keep the redundance-based strengthening steps as simple as possible. Changing the objective helps with that, since redundance-based strengthening requires showing that for an objective f and a substitution ω the constraint $f \geq f\restriction_\omega$ can be derived. Additionally, the objective update rule makes it possible for the certificate to follow the reasoning performed by presolvers more closely.

We show two ways to specify the objective change. The first approach is by stating the new objective, which is efficient when the new objective is small. The second approach is by stating the difference between the new and old objective, which is efficient for small changes to the objective.

We implemented our certification approach in PaPILO and checked the proofs using VeriPB. To evaluate our approach, we conducted experiments on MIPLIB instances converted to 0–1 ILPs [Dev20a] and the instances of the pseudo-Boolean competition 2016 [Pse16]. We experimentally verified that our approach is correct and that the overhead for generating the proof is negligible. The performance for checking the certificate could be improved. A reason for the poor checking performance is that the presolver writes preconstructed proof artefacts to the certificate, whereas the checker actually has to check the correctness of these steps. For techniques relying on propagation with compare certification using RUP against cutting planes reasoning concluding that RUP should be preferred due to smaller certificates.

5.5 Summary of Paper V

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, Febuary 2024.

Certifying algorithms move the trust that the implementation is correct from the solver to the checker. For complex systems like DRAT or VeriPB, correctness is not trivial to guarantee. The common way to address this issue is with a formally verified checker, as this reduces trust base significantly. For instance, for checking SAT solver certificates there are cake_lpr [THM23] and gratchk [Lam20]. This paper introduces the first formally verified checker for VeriPB certificates.

We introduce the formally verified checker CakePB, which is verified in the CakeML ecosystem [MO14, GMKN17]. Using the CakeML ecosystem the things that we have to trust are minimized and are either easy to check or extensively validated. This gives us the highest assurance standard for binary code extraction [KMTM18].

To assure that the certificate is certifying the correct problem, the formally verified checker must be able to parse and understand the original problem given to the solver and not just a pseudo-Boolean encoding of it. Thus, if the checker accepts the certificate, we can be sure that the answer given by the solver is correct with respect to the input problem. However, we still maintain that the checker is flexible and easily extensible to check certificates for different problems. This is achieved by separating the checker in a frontend that encodes the original problem into a pseudo-Boolean optimization problem and a backend that performs reasoning based on the pseudo-Boolean encoding. The final conclusion obtained at the end, i.e., what the certificate certified, is also translated back by the frontend to the original problem domain.

Due to the poor performance of formally verified propagation algorithms, we elaborate the proof before it is checked by CakePB, which is commonly used for SAT solving certificates [CHH+17]. The elaboration is done by VeriPB, which adds antecedents to RUP steps and syntactic implication is changed to a cutting planes derivation. We even show that elaboration can be used to synchronize slightly different encodings used in the solver and the formally verified checker.

To demonstrate that our approach works in practice, we implemented formally verified checkers for the problems of subgraph isomorphism, clique, and maximum common (connected) induced subgraph supporting all rules in the VeriPB system described in [BGMN23]. For the purpose of this thesis, the details for the specific graph problems are omitted and a detailed description of the problems can be found in [GMN20, GMM+20]. All checkers use the same backend as described above. The certifying algorithm to solve the graph problems [GMN20, GMM+20] are slightly modified to enable syncing up the different encodings. Additionally, checked deletion [BGMN23] has been fully implemented into VeriPB.

We conducted experiments on the same benchmark used in [GMN20, GMM+20]. We experimentally validated that our approach and the implementation are correct. The running time to check and elaborate the proof is slightly larger than just checking the proof. Checking the elaborated proof with CakePB is a few times faster than VeriPB, as the elaborated proof contains a lot of details that speed up the running time of the checker.

5.6 Summary of Paper VI

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.

Preprocessing is an important technique for modern MaxSAT solving [KBSJ17, IBJ22]. In this paper, we present certifying algorithms using the VeriPB system for all preprocessing techniques in the dedicated MaxSAT preprocessor MaxPre [IBJ22]. By using VeriPB we have the advantage that it is possible to integrate certification for additional techniques, like advanced symmetry breaking, in the future.

To certify standalone problem reformulation tools, like preprocessors, we extended the VeriPB system to support an output section. Hence, we certify that the problem resulting from the core constraints together with the objective at the end of the proof has the same optimal value as the original problem. Additionally, the checker verifies that the core constraints and the objective at the end of the proof match the output problem returned by the standalone reformulation tool. This is done by checking equivalence of the objective and that for every constraint in the core set there is a constraint in the output problem.

We extended the formally verified checker CakePB with support for the output section and added a frontend for MaxSAT problems. For MaxSAT preprocessing certificates, this means that we get formal guarantee that the original problem given in MaxSAT format has the same optimal value as the reformulated problem returned by the preprocessor in MaxSAT format.

We implemented certification into the MaxSAT preprocessor MaxPre. The correctness of our approach is experimentally verified using the benchmarks from the MaxSAT Evaluation 2023 [Max23]. The overhead for generating the certificate is a bit slower than desired, but a lot of time is spent on renaming variables required to match the MaxSAT format, which could be improved by adding a new rule. Most of the time for checking the proof is spent in VeriPB. Hence, we propose that the performance of the toolchain could be improved if the preprocessor would already produce RUP steps with hints.

5.7 Further Contributions Outside Included Papers

Especially with respect to VeriPB, there have been contributions that have not ended up in any publications included in the thesis. These contributions are not published in any peer-reviewed document.

First and foremost, there has been some work to make the checker ready to be used in the SAT competition and the pseudo-Boolean competition, which require a toolchain that checks the proof against a formally verified checker. For the SAT competition 2023 [BHI$^+$23], we pioneered the formal verification toolchain for decision instances, which was later published in Paper V. To improve the performance of the toolchain for the SAT competition 2024 [HIJS24], we developed RUP steps with hints similar to the hints in RUP in the LRAT format [CHH+17] and added more autoproving techniques in CakePB to reduce the amount of detail added in the elaboration. For the pseudo-Boolean competition 2024 [Pse24], we added elaboration for the advanced autoproving technique of substituted database implication, which was required by some pseudo-Boolean solver.

To avoid the synchronization of the encoding between CakePB and the checker for more elaborate encoding and to simplify the implication of certification for classical optimal planning algorithms [DHN+25], constraint labels were added to the VeriPB system.

6 Conclusions and Future Work

This thesis shows how pseudo-Boolean reasoning can be used to get efficient certifying algorithms for different combinatorial optimization paradigms. Most notably, we demonstrate how to certify state-of-the-art MaxSAT algorithms that use solution-improving and core-guided search in Papers I to III. Specifically, Paper I introduces a general approach to certify CNF encodings of pseudo-Boolean constraints used everywhere in MaxSAT solving to handle the pseudo-Boolean objective function.

We also present certification for state-of-the-art preprocessing (aka. presolving) techniques, which are crucial for the performance modern combinatorial optimization solvers. The certification approach is demonstrated for all MIP presolving techniques that preserve the variable domain $\{0, 1\}$ (see Paper IV) and for all techniques used in MaxSAT preprocessing (see Paper VI).

To guarantee that certificates generated in our VeriPB format can be trusted, we developed a formally verified proof checker in Papers V and VI that has full support for our proof system. With the approach of formal verification, the amount of code that has to be trusted is minimized and parts of the code are audited independently. However, the performance of formally verified software can not compete with untrusted software, which makes it impossible to implement some syntactic sugar rules that the untrusted checker supports. To bridge this gap, the untrusted checker has been extended with elaboration to translate the syntactic sugar to other rules that the formally verified checker supports.

6.1 Short Term Future Work

This thesis introduction is concluded with a discussion of future work on certifying combinatorial optimization based on pseudo-Boolean reasoning. We start with a discussion of short term future work that I might work on during the rest of my PhD.

First and foremost, as mentioned multiple times in the summary of the included papers, the performance of the unverified proof checker VeriPB and pipeline with the formally verified checker CakePB could be improved. Some performance issues can be resolved by spending more time on engineering the checker and using better known algorithms. For instance, the proof checker VeriPB is currently implemented in Python and C++, where Python is used for the high-level structure and C++ for low-level performance implementation improvements. However, it was observed multiple times that the use of Python and the interfacing between both languages causes a loss in performance. Therefore, the performance of the proof checker VeriPB could be improved greatly if it were implemented in one language that allows low-level implementation improvements. There has also been some research on more efficient algorithms for checking some rules, e.g., for unit propagation [NORZ24], that could be implemented for the VeriPB.

However, there are also open problems for algorithms to improve the performance of the checker and investigate how proof can be efficiently elaborated to a proof that can be checked by a formally verified checker. Specifically, the elaboration algorithm to generate the hints for reverse unit propagation is currently very naive, which makes VeriPB fast, but fewer hints might be sufficient, which might improve the performance of CakePB. It is unclear if spending more time on elaboration can improve the running time of the formally verified checking pipeline as a whole.

While it was shown by [BGMN23] that certification for fully general symmetry breaking is possible using VeriPB, it was recently discovered that their certification approach has a linear factor overhead compared to the best algorithms for logging. It should also be possible to get rid of this linear factor overhead when checking symmetry breaking certificates. The main issue is the definition of the order used for dominance-based strengthening, which requires quadratically many bits to encode. However, by using additional variables that are exclusive to the order definition, it should be possible to encode the order with linearly many bits.

To demonstrate that our proof logging approach is general, it should be possible certify incremental solving. Incremental solvers use information derived from previous calls to the solving engine to speed up subsequent calls, where it is not trivial which derivations can be reused. While there has been previous work on certifying incremental SAT solvers [FPFB24], incremental MaxSAT and pseudo-Boolean solvers are getting popular and are not certifying.

Another combinatorial problem that could be certified using pseudo-Boolean reasoning, is model (solution) enumeration and counting. The goal of model enumeration is to find all feasible solutions with respect to a set of constraints. However, in model counting we are only interested in the number of feasible solutions. There are several competing proof systems and certification approaches for counting the number of solutions satisfying a SAT formula [Cap19, FHR22, BHS23, BNAH23, CCS24]. The ideas from these systems could be generalized to certify model counting over pseudo-Boolean constraints. For model enumeration, it is still unclear what should be certified. The natural idea would be to check that the solutions satisfy all constraints and are disjoint. This would then be accompanied by a proof that shows that all solutions have been enumerated.

To improve the performance of SAT proof checkers, proofs are checked starting with the conclusion and only the steps required to show the conclusion are checked [WHH14], which can significantly reduce the number of rules that have to be checked. The checker can then also produce a trimmed proof that only contains the steps required to show the conclusion. However, implementing this idea for our proof system is a non-trivial task due to the redundance-based strengthening rules and its proof obligations.

The idea of weak substitution redundancy could be added to the redundance-based strengthening rule to enable more expressive reasoning to achieve shorter proofs.

6.2 Long Term Future Work

Finally, we discuss some future long term directions that research in certifying algorithms can take.

The current proof system operates with pseudo-Boolean constraint, but it should be possible to generalize the rules to constraints that use rational coefficients and use integer or even rational variables. There already exists preliminary work for such an extension of the proof system [DEGH23], but there are still unclear how efficient logging and checking can be implemented for this theoretical proof system.

While parallel and distributed combinatorial solvers are becoming more and more mainstream [SRB25], almost all checkers are currently completely sequential. There are several ways to divide the checking between different cores and machines. For distributed checking the proof could be divided into consecutive parts and each part is checked independently on one machine, but this requires knowing the database at the start of each part. Another idea that is more viable in a parallel setting is checking each rule in parallel. This would require that the database can be accessed asynchronously.

As proof checking becomes comparable to the performance of solvers [PFB23, Lam24], it becomes viable to run a proof checker in parallel to a solver. This would reduce the time to receive a trusted result for the problem and the solver immediately fails when its reasoning is incorrect. To reduce the overhead for reading and writing the proof, the data structure for each rule could directly be constructed by the solver and then send to the checker through an application programming interface.

To make it as easy as possible to make certification mainstream and implemented in more and more solvers, it might help to have a library that implements certification for common reasoning techniques. This should be very helpful, as we observed that similar reasoning is used in all kinds of solvers. Alternatively, the proof format could also be made extensible so that new rules can be defined in terms of existing rules, which makes it possible to replace these new rules with standard VeriPB rules in the elaboration phase.

VeriPB currently only has one fixed mode for checking the proof on how strict it is with the claimed reasoning being correct. However, different applications might require different modes. If we are only interested in knowing that the result is correct, we could use a permissive mode that tries to fix up the proof if the reasoning is slightly wrong. The other extreme could be a strict mode where VeriPB enforces that all reasoning in the proof is correct without any ambiguities, which can be helpful for debugging and finding bugs as fast as possible.

The permissive mode and the autoproving performed by VeriPB in general could be made even more powerful by integrating a pseudo-Boolean solver, e.g., RoundingSat. The idea would be to call a pseudo-Boolean solver on the formula and the negation of the constraint that we want to derive. We consider this approach to be successful if the solver derives contradiction. This can be compared to Sledgehammer [BN10] for the higher-order logic proof assistant Isabelle [NWP02]. However, allowing a pseudo-Boolean solver to check the correctness of a rule breaks the guarantee each proof rule is checkable in polynomial time.

For developing and prototyping it might be beneficial to have an interactive mode in the checker. This could mean that the proof checker runs until a specified point in the proof and then a user can interact with the checker using the rules of the proof system. Similar to a debugger the user might also be able to explore the current state of the checker by viewing the constraints available at the given point in the proof.

Finally, proofs can also be used to analyse the reasoning performed by solvers to better understand which techniques and heuristics are beneficial for making progress to the result. This might also unveil performance bugs due to suboptimal reasoning, e.g., a derived contradiction has a slack that is larger than one with respect to the empty assignment. The analysis can also be used to explain why the solver came to its result in a format that is understandable to humans.

References

[AB16] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, 4th edition, 2016.

[ABG$^+$20] Tobias Achterberg, Robert E. Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32(2):473–506, 2020.

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABS13] Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, pages 39–55, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Ach07] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007. Available at `https://opus4.kobv.de/opus4-zib/files/1112/Achterberg_Constraint_Integer_Programming.pdf`.

[AGJ+18] 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.

[AH14] André Abramé and Djamal Habet. ahmaxsat: Description and evaluation of a branch and bound max-sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):89–128, 2014.

[AKMS12] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP '12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 211–221, September 2012.

[ALM09] Josep Argelich, Inês Lynce, and João P. Marques-Silva. On solving Boolean multilevel optimization problems. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 393–398, July 2009.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS '68)*, volume 32, pages 307–314, April 1968.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BBL24] Ilario Bonacina, Maria Luisa Bonet, and Massimo Lauria. MaxSAT Resolution with Inclusion Redundancy. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, volume 305 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:15, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BBN$^+$23] 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.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BBVC13] Felix Brandt, Reinhard Bauer, Markus Völker, and Andreas Cardeneo. A constraint programming-based approach to a large-scale energy management problem with varied constraints: A solution approach to the roadef/euro challenge 2010. *Journal of Scheduling*, 16(6):629–648, 2013.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BFF$^+$24] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions*, volume B-2024-1 of *Department of Computer Science Report Series B*, pages 8–10. University of Helsinki, 2024.

[BFT11] Frédéric Besson, Pascal Fontaine, and Laurent Théry. A Flexible Proof Format for SMT: a Proposal. In Pascal Fontaine and Aaron Stump, editors, *First International Workshop on Proof eXchange for Theorem Proving - PxTP 2011*, Wroclaw, Poland, August 2011.

[BGMN23] 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*.

[BHI$^+$23] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

[BHS23] Olaf Beyersdorff, Tim Hoffmann, and Luc Nicolas Spachmann. Proof Complexity of Propositional Model Counting. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[BJ17] Jeremias Berg and Matti Järvisalo. 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)*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, August 2017.

[BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Biere et al. [BHvMW21], chapter 9, pages 391–435.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Biere et al. [BHvMW21], chapter 24, pages 929–991.

[Bla37] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BN10] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BNAH23] Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. 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 6:1–6:20, July 2023.

[BT21] Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17(2):12:1–12:31, April 2021. Preliminary version in *SAT '19*.

[Cap19] Florent Capelli. Knowledge compilation languages as proof systems. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 90–99, Cham, 2019. Springer International Publishing.

[CCS24] Sravanthi Chede, Leroy Chew, and Anil Shukla. Circuits, Proofs and Propositional Model Counting. In Siddharth Barman and Sławomir Lasota, editors, *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2024)*, volume 323 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:23, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CGS17] 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.

[CHH$^+$17] 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.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CNR21] Aviad Cohen, Alexander Nadel, and Vadim Ryvchin. Local search with a sat oracle for combinatorial optimization. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 87–104, Cham, 2021. Springer International Publishing.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, May 1971.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979. Preliminary version in *STOC '74*.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[Dev20a] Jo Devriendt. Miplib 0-1 instances in opb format. Dataset on Zenodo, 05 2020.

[Dev20b] Jo Devriendt. Watched propagation of 0-1 integer linear constraints. 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 160–176. Springer, September 2020.

[DG02] Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI '02)*, pages 635–640, July 2002.

[DGD+21] 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.

[DHN+25] 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)*, November 2025. To appear.

[Die16] Reinhard Diestel. *Graph Theory*. Springer-Verlag, Heidelberg (print edition), 2016.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DMM$^+$24] 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.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[EGMN20] 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.

[EN18] 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.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560, July 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[Far02] Julius Farkas. Theorie der einfachen Ungleichungen. *Journal für die Reine und Angewandte Mathematik*, 1902(124):1–27, 1902.

[FHR22] Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, pages 30:1–30:24, 2022.

[FL23] Mathias Fleury and Peter Lammich. A more pragmatic CDCL for IsaSAT and targetting LLVM (short paper). In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29)*, volume 14132 of *Lecture Notes in Computer Science*, pages 207–219. Springer, July 2023.

[FPFB24] Katalin Fazekas, Florian Pollitt, Mathias Fleury, and Armin Biere. Certifying incremental sat solving. In *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius*, volume 100, pages 321–340, 2024.

[FSM$^+$24] 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 Paul Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, volume 307 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[FYBH24] Nils Froleyks, Emily Yu, Armin Biere, and Keijo Heljanko. Certifying phase abstraction. In *International Joint Conference on Automated Reasoning*, pages 284–303. Springer, 2024.

[GGH22] Ambros Gleixner, Leona Gottwald, and Alexander Hoen. PaPILO: A parallel presolving library for integer and linear programming with multiprecision support. Technical Report 2206.10709, arXiv.org, June 2022.

[GGK$^+$19] Gerald Gamrath, Ambros Gleixner, Thorsten Koch, Matthias Miltenberger, Dimitri Kniasew, Dominik Schlögel, Alexander Martin, and Dieter Weninger. Tackling industrial-scale supply chain problems by mixed-integer programming. *Journal of Computational Mathematics*, 37(6):866–888, 2019.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM$^+$20] 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.

[GMN20] 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.

[GMN22] 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.

[GN03] 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.

[GN21] 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.

[GN22] Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. Technical Report 2209.12185, arXiv.org, September 2022.

[GNY19] Stephan Gocht, Jakob Nordström, and Amir Yehudayoff. On division versus saturation in pseudo-Boolean solving. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI '19)*, pages 1711–1718, August 2019.

[Goc22] 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`.

[GS19] Graeme Gange and Peter Stuckey. Certifying optimality in constraint programming. Presentation at KTH Royal Institute of Technology, February 2019.

[GSD19] 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.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.

[Heu21] Marijn JH Heule. Proofs of unsatisfiability. In *Handbook of Satisfiability*, pages 635–668. IOS Press, 2021.

[HGH23] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective Auxiliary Variables via Structured Reencoding. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference* *on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HHW14] Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014.

[HIJS24] Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2024.

[HKB17] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, August 2017.

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

[HL06] Federico Heras and Javier Larrosa. New inference rules for efficient max-sat solving. In *AAAI*, pages 68–73, 2006.

[HPRS24] Roghayeh Hajizadeh, Tatiana Polishchuk, Elina Rönnberg, and Christiane Schmidt. A dantzig-wolfe reformulation for automated aircraft arrival scheduling in tmas. In *Proceedings of the 14th International Conference on the Practice and Theory of Automated Timetabling, PATAT 2024 :*, pages 268–271, 2024.

[HT15] Osman Hasan and Sofiene Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global Scientific Publishing, 2015.

[IBJ21] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Refined core relaxation for core-guided maxsat solving. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, pages 28–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings* *of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João P. Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, September 2019.

[JBBJ25] Christoph Jabs, Jeremias Berg, Bart Bogaerts, and Matti Järvisalo. Certifying pareto-optimality in multi objective maximum satisfiability. In Arie Gurfinkel and Marijn Heule, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–129, Cham, 2025. Springer Nature Switzerland.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KBSJ17] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: an extended maxsat preprocessor. In Serge Gaspers and Toby Walsh, editors, *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing, (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017.

[KLM$^+$25] 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)*, August 2025. To appear.

[KMMS06] Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM Journal on Computing*, 36(2):326–353, 2006.

[KMTM18] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, volume 10895 of *Lecture Notes in Computer Science*, pages 362–369. Springer, July 2018.

[Kra19] Jan Krajíček. *Proof Complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications*. Cambridge University Press, March 2019.

[KRH18] Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, volume 10900 of *Lecture Notes in Computer Science*, pages 516–531. Springer, July 2018.

[KT24] Leszek Kołodziejczyk and Neil Thapen. The strength of the dominance rule. 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 20:1–20:22, August 2024.

[KZFH12] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver: System description. *Journal on Satisfiability, Boolean Modelling and Computation*, 8(1-2):95–100, 2012.

[Lam20] 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.

[Lam24] Peter Lammich. Fast and verified unsat certificate checking. In Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning*, pages 439–457, Cham, 2024. Springer Nature Switzerland.

[Lev73] Leonid A. Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973. In Russian. Available at `http://mi.mathnet.ru/ppi914`.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[LXC$^+$21] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for MaxSAT. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:18, October 2021.

[MAGL11] João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[Mar10] João P. Marques-Silva. Minimal unsatisfiability: Models, algorithms and applications (Invited paper). In *Proceedings of the 40th IEEE International Symposium on Multiple-Valued Logic*, pages 9–14, May 2010.

[Max22] MaxSAT evaluation 2022. `https://maxsat-evaluations.github.io/2022`, August 2022.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MDM14] António Morgado, Carmine Dodaro, and João P. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, September 2014.

[MJML14] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, September 2014.

[MKL$^+$95] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[MM23] 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.

[MM25] 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.

[MML14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. 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 438–445. Springer, July 2014.

[MMN24] 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.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MN89] Kurt Mehlhorn and Stefan Näher. Leda a library of efficient data types and algorithms. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, pages 88–106, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[MN95] Kurt Mehlhorn and Stefan Näher. Leda: a platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, January 1995.

[MO12] David F. Manlove and Gregg O'Malley. Paired and altruistic kidney donation in the UK: Algorithms and experimentation. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA '12)*, volume 7276 of *Lecture Notes in Computer Science*, pages 271–282. Springer, June 2012.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[MOS15] Renata Mansini, Włodzimierz Ogryczak, and M. Grazia Speranza. *Linear and mixed integer programming for portfolio optimization*, volume 21. Springer, 2015.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[MSLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Biere et al. [BHvMW21], chapter 4, pages 133–182.

[NB14] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[NORZ24] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Rui Zhao. Speeding up pseudo-Boolean propagation. 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 22:1–22:18, August 2024.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[PB23] Tobias Paxian and Armin Biere. Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, July 2023.

[PFB23] Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. 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 21:1–21:12, July 2023.

[PR16] Tobias Philipp and Adrián Rebola-Pardo. DRAT proofs for XOR reasoning. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429. Springer, November 2016.

[PRB18] 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.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, January 2021.

[PS15] Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-Boolean constraints into CNF. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, September 2015.

[Pse16] Pseudo-Boolean competition 2016. `https://www.cril.univ-artois.fr/PB16/`, July 2016.

[Pse24] Pseudo-Boolean competition 2024. `https://www.cril.univ-artois.fr/PB24/`, August 2024.

[RM21] Olivier Roussel and Vasco Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [BHvMW21], chapter 28, pages 1087–1129.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RP23] Adrián Rebola-Pardo. Even Shorter Proofs Without New Variables. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, volume 271 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[RS18] Adrián Rebola-Pardo and Martin Suda. A theory of satisfiability-preserving proofs in sat solving. In *LPAR*, pages 583–603, 2018.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[S$^+$20] Philine Schiewe et al. *Integrated optimization in public transport planning*, volume 160. Springer, 2020.

[Sau24] J. Sauppe, editor. *Mathematical Programming Glossary*. INFORMS Computing Society, `http://glossary.informs.org`, 2006–24. Originally authored by Harvey J. Greenberg, 1999-2006.

[Sch05] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, pages 1–68. Elsevier, 2005.

[Sch12] Verena Schmid. Solving the dynamic ambulance relocation and dispatching problem using approximate dynamic programming. *European Journal of Operational Research*, 219(3):611–621, 2012. Feature Clusters.

[SH23] Bernardo Subercaseaux and Marijn J. H. Heule. The packing chromatic number of the infinite square grid is 15. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–406, Cham, 2023. Springer Nature Switzerland.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[SN15] Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.

[SRB25] Dominik Schreiber, Niccolò Rigi-Luperti, and Armin Biere. Streamlining Distributed SAT Solver Design. In Jeremias Berg and Jakob Nordström, editors, *28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025)*, volume 341 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:23, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[TD20] Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. On certifying the UNSAT result of dynamic symmetry-handling-based SAT solvers. *Constraints*, 25(3–4):251–279, December 2020.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.

[Van08] 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`.

[VDB22] 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.

[VG02] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *7th International Symposium on AI and Mathematics*, 2002.

[VS10] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, pages 204–209, July 2010.

[Wal96] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1):139–168, 1996.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, October 1998.

[Wei25] Nils Weidmann. Multi-League Sports Scheduling with Team Interdependencies: An Optimization Model. In Maria Garcia de la Banda, editor, *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[WHH14] 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.

[WS24] Dominik Winterer and Zhendong Su. Validating smt solvers for correctness and performance via grammar-based enumeration. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[YBH21] Emily Yu, Armin Biere, and Keijo Heljanko. Progress in certifying hardware model checking results. In *International Conference on Computer Aided Verification*, pages 363–386. Springer, 2021.

[ZWCX22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022.

Included Papers

Certified CNF Translations for Pseudo-Boolean Solving

Abstract

The dramatic improvements in Boolean satisfiability (SAT) solving since the turn of the millennium have made it possible to leverage state-of-the-art conflict-driven clause learning (CDCL) solvers for many combinatorial problems in academia and industry, and the use of proof logging has played a crucial role in increasing the confidence that the results these solvers produce are correct. However, the fact that SAT proof logging is performed in conjunctive normal form (CNF) clausal format means that it has not been possible to extend guarantees of correctness to the use of SAT solvers for more expressive combinatorial paradigms, where the first step is an unverified translation of the input to CNF.

In this work, we show how cutting-planes-based reasoning can provide proof logging for solvers that translate pseudo-Boolean (a.k.a. 0-1 integer linear) decision problems to CNF and then run CDCL. To support a wide range of encodings, we provide a uniform and easily extensible framework for proof logging of CNF translations. We are hopeful that this is just a first step towards providing a unified proof logging approach that will also extend to maximum satisfiability (MaxSAT) solving and pseudo-Boolean optimization in general.

1 Introduction

Boolean satisfiability (SAT) solving has witnessed striking improvements over the last couple of decades, starting with the introduction of *conflict-driven clause learning (CDCL)* SAT solvers [MS99, MMZ$^+$01], and this has led to a wide range of applications including large-scale problems in both academia and industry [BHvMW21]. The conflict-driven paradigm has also been successfully exported to other areas such

Stephan Gocht, Jakob Nordström, Ruben Martins, and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". Accepted for publication in *Journal of Artificial Intelligence Research*. Preliminary version 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.

as *maximum satisfiability (MaxSAT)*, *pseudo-Boolean (PB) solving*, *constraint programming (CP)*, and *mixed integer linear programming (MIP)*. As modern combinatorial solvers are used to attack ever more challenging problems, and employ ever more sophisticated heuristics and optimizations to do so, the question arises whether we can trust the results they produce. Sadly, it is well documented that state-of-the-art CP and MIP solvers can return incorrect solutions [AGJ$^+$18, CKSW13, GSD19]. For SAT solvers, however, analogous problems [BLB10] have been successfully addressed by the introduction of *proof logging*, requiring that solvers should be *certifying* [MMNS11] in the sense that they output machine-verifiable proofs of their claims that can be verified by a stand-alone *proof checker*.

A number of different proof logging formats have been developed for SAT solving, including *RUP* [GN03, Van08], *TraceCheck* [Bie06], *DRAT* [HHW13a, HHW13b, WHH14], *GRIT* [CFMSSK17], and *LRAT* [CFHH$^+$17]. Since 2013, the SAT competitions [BBHJ13] require solvers to be certifying, with *DRAT* established as the standard format. It would be highly desirable to have such proof logging also for stronger combinatorial solving paradigms, but while methods such as *DRAT* are extremely powerful in theory, the limitation to a clausal format makes it hard to capture more advanced forms of reasoning in a succinct way. A more fundamental concern is that it is not clear how these proof logging methods should deal with input that is not presented in conjunctive normal form (CNF). One way to address this problem could be to allow extensions to the *DRAT* format [BCH21]. However, we focus on another approach pursued in recent years to develop stronger proof logging methods based on more expressive formalisms such as binary decision diagrams [BB21], algebraic reasoning [KBBN22, KB21, KFB20, RBK$^+$18], pseudo-Boolean reasoning [EGMN20, GMM$^+$20, GMN20, GN21, BGMN22, GMN22], and integer linear programming [CGS17, EG21].

Our Contribution In this work, we consider the use of CDCL for pseudo-Boolean solving, where the pseudo-Boolean input (i.e., a 0-1 integer linear program) is translated to CNF and passed to a SAT solver, as pioneered in MiniSat+ [ES06]. The two solvers NaPS [SN15] and Open-WBO [MML14] using this approach were among the top performers in the latest pseudo-Boolean evaluation in 2016. While *DRAT* proof logging can certify unsatisfiability of the translated formula, it cannot prove correctness of the translation, not only since there is no known method of carrying out PB reasoning efficiently in *DRAT* (except for constraints with small coefficients [BBH22]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the *cutting planes* proof method [CCT87], enhanced with a rule for introducing extension variables [GN21], to show that the CNF formula resulting from the translation can be derived from the original pseudo-Boolean constraints. Since this method is a strict extension of *DRAT*, we can combine the proof for the translation with the SAT solver *DRAT* proof log (with appropriate syntactic modifications). In this way we achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker

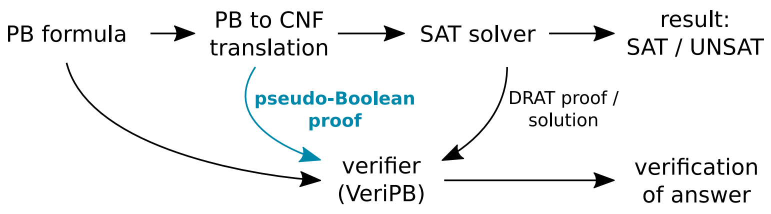


Figure 1: *Proof logging workflow for pseudo-Boolean solving with our contribution highlighted in blue boldface.*

VeriPB [GN21, BGMN22] as illustrated in Figure 1. We note that verifying the correctness of the pseudo-Boolean encoding for the problem is beyond the scope of this paper.

One challenge when certifying PB-to-CNF translations is that there are many different ways of encoding pseudo-Boolean constraints into CNF (as catalogued in, e.g., [PS15]), and it is time-consuming (and error-prone) to code up proof logging for every single encoding. However, many of the encodings can be understood as first designing a circuit to evaluate whether the PB constraint is satisfied, and then writing down a CNF formula enforcing the computation of this circuit. An important part of our contribution is that we develop a general proof logging method for a wide class of such circuits. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the circuit computations, and once this has been done the desired clauses in the CNF translation can simply be obtained by so-called *reverse unit propagation (RUP)* [GN03, Van08], obviating the need for complicated syntactic proofs. We apply this method to the *sequential counter* [Sin05], *totalizer* [BB03], *generalized totalizer* [JMM15] and *binary adder network* [ES06, War98] encodings, and report results from an empirical evaluation of the efficiency of proof generation and verification. As an additional application, we show how our certified PB-to-CNF translations can be combined with SAT proof logging to certify, for the first time, the correctness of claimed optimal values for instances in the MaxSAT Evaluation 2022.

We note that a stronger result than certifying that the CNF translation can be derived from the pseudo-Boolean input would be to certify *equivalence* of the original pseudo-Boolean formula F and the translated CNF formula F', in the sense that (a) any satisfying assignment α to F could be extended to an assignment α' also to the new variables introduced during translation that would satisfy F', and that (b) any satisfying assignment α' to F' also has to satisfiy F. The tools we develop can reach this more ambitious goal in principle, but since some additional technical problems arise along the way we have to leave this as future work.

Outline of This Paper After discussing preliminaries in Section 2, we illustrate our method for the sequential counter encoding in Section 3. Section 4 presents the general framework, and we discuss how to apply it to adder networks in

Section 5 and (generalized) totalizer encoding in Section 6. We report data from our experimental evaluation in Section 7 and conclude with a discussion of some directions for future research in Section 8.

2 Preliminaries

Let us start with a review of some standard material that can also be found in, e.g., [BN21, GN21]. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x}$, where variables can be assigned values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. For notational convenience, we define $\overline{\overline{x}} \doteq x$ (where we use $\doteq$ to denote syntactic equality). We write $[n] = \{1, 2, \ldots, n\}$ to denote the n first positive integers, and sometimes write $\vec{x} = \{x_i \mid i \in [n]\}$ to denote a set of variables, where the size n of the set is understood from context (or is not important). A *pseudo-Boolean (PB) constraint* is a 0-1 linear inequality

$$C \doteq \textstyle\sum_i a_i \ell_i \geq A\,, \tag{1}$$

which without loss of generality we always assume to be in *normalized form* [Bar95]; 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 normalized form of the *negation* of C in (1) is the constraint

$$\neg C \doteq \textstyle\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1 \tag{2}$$

(encoding that the sum of the coefficients of falsified literals in C is so large that coefficients of satisfied literals can contribute at most $A - 1$). We use equality constraints

$$C \doteq \textstyle\sum_i a_i \ell_i = A \tag{3a}$$

as syntactic sugar for the pair of inequalities

$$C^{\Rightarrow} \doteq \textstyle\sum_i a_i \ell_i \geq A \tag{3b}$$

and

$$C^{\Leftarrow} \doteq \textstyle\sum_i -a_i \ell_i \geq -A \tag{3c}$$

(with the latter converted to normalized form). We write $\sum_i a_i \ell_i \bowtie A$ for $\bowtie \in \{\geq, \leq, =\}$ for constraints that are either inequalities or equalities. A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints. A *cardinality constraint* is a PB constraint with all coefficients equal to 1. If the degree is also 1, then the constraint

$$\ell_1 + \cdots + \ell_k \geq 1 \tag{4a}$$

is equivalent to the *(disjunctive) clause*

$$\ell_1 \vee \cdots \vee \ell_k\,, \tag{4b}$$

and so CNF formulas are just special cases of pseudo-Boolean 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 as in (1) yields the constraint $C\!\restriction_\rho$ obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, and for a formula F we define $F\!\restriction_\rho = \bigwedge_j C_j\!\restriction_\rho$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$ (or, equivalently, if the restricted constraint $C\!\restriction_\rho$ has a non-positive degree and is thus trivial). An assignment ρ satisfies $F \doteq \bigwedge_j C_j$ if it satisfies all C_j, in which case F is *satisfiable*. A formula without satisfying assignments is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Cutting planes as defined in [CCT87] is a method for iteratively deriving new constraints C implied by a PB formula F. If C and D are previously derived constraints, or are *axiom constraints* in F, then any positive integer *linear combination* of these constraints can be derived. (By a linear combination of two equality constraints C and D, we mean the identical linear combinations of $C^{\Rightarrow}$ and $D^{\Rightarrow}$ and of $C^{\Leftarrow}$ and $D^{\Leftarrow}$, respectively.) We can also add *literal axioms* $\ell_i \geq 0$ to a previously derived constraint. For a constraint $\sum_i a_i \cdot \ell_i \geq A$ in normalized form, we can use *division* by a positive integer d to derive $\sum_i \lceil a_i/d \rceil \ell_i \geq \lceil A/d \rceil$, dividing and rounding up the degree and coefficients, and it is sometimes convenient to also include a *saturation* rule deriving $\sum_i \min\{a_i, A\} \cdot \ell_i \geq A$ from $\sum_i a_i \cdot \ell_i \geq A$. We remark that the soundness of the division and saturation rules as stated depends on the constraints being presented in normalized form.

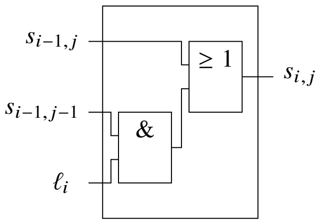
For PB formulas F, F' and constraints C, C', we say that F *implies* or *models* C, denoted $F \models C$, if any assignment satisfying F must also satisfy C, and we write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is clear that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense, and cutting planes is an *implicationally complete* method in the sense that any implied constraint can also be derived syntactically.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C\!\restriction_\rho$ cannot be satisfied unless ℓ is set to true. During *unit propagation* on F under ρ, we extend ρ iteratively by assignments to any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which some constraint C propagates a literal that has already been assigned to the opposite value. The latter scenario is called a *conflict*, since ρ' *violates* the constraint C in this case. 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, but the opposite direction is not necessarily true.

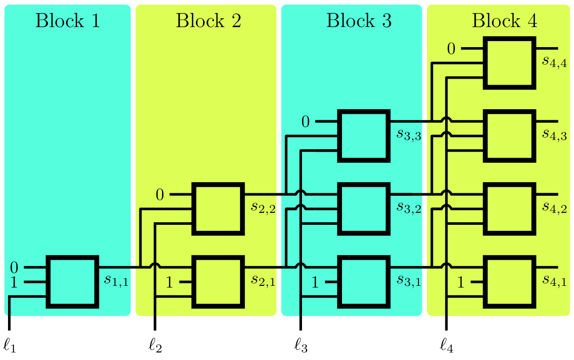
For introducing new variables, we will use the *reification* rule saying that we can introduce the *reified constraints*

$$z \Rightarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad A\overline{z} + \sum_i a_i \ell_i \geq A \tag{5a}$$

$$z \Leftarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad \left(\sum_i a_i - A + 1\right) \cdot z + \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1 \tag{5b}$$



(a) *Logic circuit of single component.*



(b) *Circuit for* 4 *input literals counting up to* 4.

Figure 2: *Circuit representation of the sequential counter encoding.*

provided that z is a *fresh variable* that is not in the formula and has not appeared previously in the derivation. A moment of thought reveals that the constraint (5a) says that if z is true, then $\sum_i a_i\ell_i \geq A$ has to hold, and this explains the notation $z \Rightarrow \sum_i a_i\ell_i \geq A$ introduced for this constraint. In an analogous fashion, the constraint (5b) says that if $\sum_i a_i\ell_i \geq A$ holds, then z has to be true. We will write $z \Leftrightarrow \sum_i a_i\ell_i \geq A$ for the conjunction of the constraints (5a) and (5b). Adding such reification constraints preserves equisatisfiability, since any satisfying assignment to F can be extended by setting the fresh variable z as required to satisfy the implications. The reification rule is a special case of the redundance rule in [GN21], where we can add any *redundant* constraint C with the property that F and $F \wedge D$ are equisatisfiable.

3 Certified CNF Translation Using the Sequential Counter Encoding

To give a concrete illustration of our approach for proving the correctness of translations of pseudo-Boolean constraints, in this section we consider how to convert cardinality constraints $\sum_{i=1}^{n} \ell_i \bowtie k$ to CNF using the *sequential counter* encoding [Sin05]. This encoding is based on a circuit summing up the input bits one by one, with intermediate variables $s_{i,j}$ for $i \in [n]$ and $j \in [i]$ evaluating to true if and only if $\sum_{t=1}^{i} \ell_t \geq j$ holds. The variables $s_{i,j}$ can be computed inductively as in Figure 2a by the formula

$$s_{i,j} \leftrightarrow \big((\ell_i \wedge s_{i-1,j-1}) \vee s_{i-1,j}\big) \tag{6}$$

saying that $s_{i,j}$ is true either if the first $i-1$ literals add up to $j-1$ and the ith literal is true, or if already the first $i-1$ literals add up to j. The circuit constructed in this way, shown in Figure 2b, can be partitioned into n blocks, where the ith block computes the variables $s_{i,j}$ for $j \in [i]$ from the ith input bit ℓ_i and the variables $s_{i-1,j}$ in the previous block. Identifying such blocks in the circuit is a key component in our method for proving that the CNF translation is correct.

For the sequential counter circuit, we obtain the CNF encoding of the constraint $\sum_{i=1}^{n} \ell_i \bowtie k$ by translating each component in Figure 2a (as described by Equation (6)) to the clausal constraints

$$\bar{\ell}_i + \bar{s}_{i-1,j-1} + s_{i,j} \geq 1 \tag{7a}$$
$$\bar{s}_{i-1,j} + s_{i,j} \geq 1 \tag{7b}$$
$$\ell_i + s_{i-1,j} + \bar{s}_{i,j} \geq 1 \tag{7c}$$
$$s_{i-1,j-1} + \bar{s}_{i,j} \geq 1 \tag{7d}$$

for $i \in [n]$ and $j \in [i]$. For all i we set $s_{i,0} = 1$ and simplify, so that constraint (7a) turns into $\bar{\ell}_i + s_{i,1} \geq 1$ and constraint (7d) is satisfied and disappears. We also set $s_{i-1,i} = 0$, so that (7c) becomes $\ell_i + \bar{s}_{i,i} \geq 1$ and (7b) is satisfied and disappears.

Once clauses (7a)–(7d) have been generated for all circuit components, we obtain a greater-than-or-equal-to-k constraint by adding the unit clause $s_{n,k} \geq 1$. Analogously, a less-than-or-equal-to-k constraint is enforced using the clause $\bar{s}_{n,k+1} \geq 1$. A common optimization, known as *k-simplification*, is to omit clauses corresponding to the computation of variables $s_{i,j}$ for $j > k+1$, as such variables are not relevant for deciding whether the cardinality constraint is true or not.

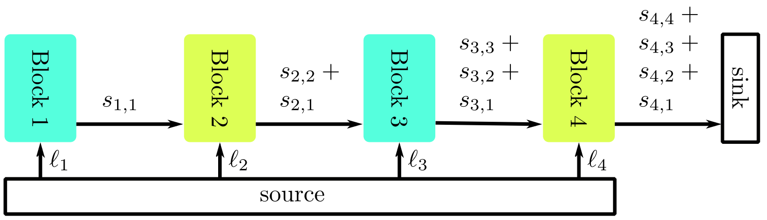
As a preparation for our proof logging discussions, let us study the variables $s_{i,j}$ in more detail, ignoring k-simplification for now. Since $s_{i,j}$ is true if and only if $\sum_{t=1}^{i} \ell_t \geq j$ holds, for all $i \in [n]$ we should be able to deduce

$$\textstyle\sum_{t=1}^{i} \ell_t = \sum_{j=1}^{i} s_{i,j} \, . \tag{8}$$

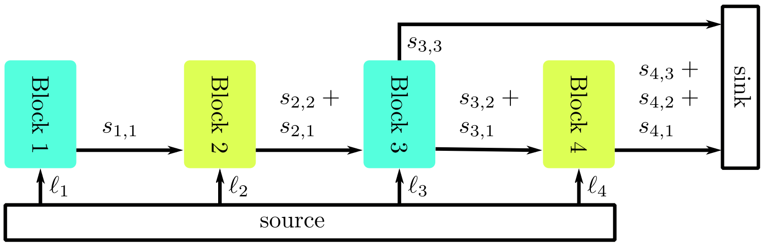
However, the sequential counter circuit computes the variables $s_{i,j}$ in the ith block using only the variables $s_{i-1,j}$ from the previous block and the literal ℓ_i, and so if we only reason locally about the ith block what we can derive is the equality

$$\textstyle\ell_i + \sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \, . \tag{9}$$

If we look at the variables on wires entering and exiting the ith block of the circuit, we see that Equation (9) specifies that the sum of the inputs is equal to the sum of the outputs. If we represent the circuit in Figure 2b as a graph with every block contracted into a single node and the literals ℓ_i in the cardinality constraint collected into another separate node, then every ith block node has an incoming edge from the literals node and (for $i > 1$) another edge from the $(i-1)$th block node, and an outgoing edge to the $(i+1)$th block node (or, for $i = n$, to a special sink node that we can also introduce). If we label the incoming edges by ℓ_i and $\sum_{j=1}^{i-1} s_{i-1,j}$ and the outgoing edge by $\sum_{j=1}^{i} s_{i,j}$, as shown in Figure 3a, then we can view (9) as saying that for all vertices in the graph the sum of the labels of input edges should be equal to the sum of the output edge. We will refer to this as a *preservation equality*.


Block 1
Block 2
Block 3
Block 4
sink
source
$s_{1,1}$
$s_{2,2}+$ $s_{2,1}$
$s_{3,3}+$ $s_{3,2}+$ $s_{3,1}$
$s_{4,4}+$ $s_{4,3}+$ $s_{4,2}+$ $s_{4,1}$
ℓ_1 ℓ_2 ℓ_3 ℓ_4

(a) *Graph without k-simplification.*



(b) *Graph with k-simplification for* $k = 1$.

Figure 3: *Graph representation of the sequential counter encoding.*

What is not at all obvious from this particular example, but what we will show in later sections, is that many CNF translations of pseudo-Boolean constraints can be represented as graphs with preservation equalities in a similar way, though sometimes with larger coefficients in the linear combinations of the literals. And, jumping ahead a bit, our main contribution in this paper is a generic proof logging method that will certify correctness for any CNF encoding that can be represented in this graph framework with preservation equalities.

Using the graph representation we can easily see that the telescoping sum of the preservation equalities for all nodes derives (8). From this, in turn, it is clear that a constraint on the input variables $\sum_{j=1}^{n}\ell_i \bowtie k$ implies the same constraint on the output variables, and formally this can be obtained by one final telescoping sum step combining $\sum_{j=1}^{n}\ell_i \bowtie k$ and $\sum_{j=1}^{n}\ell_i = \sum_{j=1}^{n}s_{n,j}$ to get

$$\sum_{j=1}^{n}s_{n,j} \bowtie k\,. \tag{10}$$

Another important property of the variables $s_{i,j}$ is that they do not just take any values satisfying (9), but are ordered—since $s_{i,j}$ encodes $\sum_{t=1}^{i}\ell_t \geq j$, it follows that $s_{i,j}$ cannot be true unless also $s_{i,j'}$ is true for all $j' < j$. This can be expressed by *ordering constraints*

$$s_{i,j} \geq s_{i,j+1} \qquad\qquad i \in [n]\,, j \in [i-1]\,, \tag{11}$$

which are semantically implied by the circuit encoding.

Taking this view of the circuit encoding, the task of certifying the correctness of the CNF translation becomes surprisingly simple. If we can derive the pseudo-Boolean constraints (9)–(11), then it can be verified that the clauses of the sequential counter encoding (i.e., (7a)–(7d) plus $\bar{s}_{n,k+1} \geq 1$ and/or $s_{n,k} \geq 1$) all follow by reverse unit propagation. This is so since when asserting the clauses to false, the ordering constraints (11) will propagate enough variables $s_{i,j}$ for (9) to be falsified.

To see how to obtain the constraints (9)–(11), note that we already discussed above how to derive (10) by a telescoping sum over constraints (9), which is straightforward to do with standard cutting planes rules. To get constraints on the form (9), we can use reification to define the meaning of the variables $s_{i,j}$ by constraints

$$s_{i,j} \Leftrightarrow \ell_i + \textstyle\sum_{j=1}^{i-1} s_{i-1,j} \geq j \tag{12}$$

(with notation as introduced in (5a)–(5b) in Section 2). If we do this in increasing order for i and j, then $s_{i,j}$ is fresh in (12) and so these are valid derivation steps. From the constraints (12) we can then derive (9) and (11) as illustrated in the next example. To show the concrete syntax used in the proof file, the example is interleaved with according proof file snippets and concatenating all the snippets would result in a full proof that can be checked using the pseudo-Boolean proof checker VeriPB [EGMN20, GN21, GMN20].

Example 1. Every constraint in the proof format is assigned a unique *identifier* (ID) and constraints that are derived in the proof are annotated in this example by their corresponding identifier.

Let us consider the constraint

$$(\text{ID: } 1) \qquad x_1 + \bar{x}_2 \geq 2 \tag{13}$$

to be encoded with the sequential counter encoding. To use this constraint as input for VeriPB, the constraint is written in the OPB format [RM16], which is extended to offer, among other thing, a greater flexibility for variable names. The input file would contain the following two lines.

```
* #variables= 2 #constraints= 1
+1 x1 +1 ~x2 >= 2 ;
```

The proof file for this instance starts with the header

```
pseudo-Boolean proof version 2.0
f 1
```

to tell the checker which version of the proof format is used and to load the formula that should contain one constraint.

The proof starts with deriving the preservation equality

$$x_1 = s_{1,1} \tag{14}$$

for the first block of the sequential counter encoding. The fresh counter variable $s_{1,1}$ is introduced by reification resulting in the constraints

$$(\text{ID: } 2) \qquad \bar{s}_{1,1} + x_1 \geq 1 \tag{15a}$$

$$(\text{ID: } 3) \qquad s_{1,1} + \bar{x}_1 \geq 1 \tag{15b}$$

to be added. In the proof log the reified constraints can be added using the redundance-based strengthening rule with the according witness.

```
red +1 ~s11 +1  x1 >= 1 ; s11 -> 0
red +1  s11 +1 ~x1 >= 1 ; s11 -> 1
```

The constraints in (15) together represent the desired preservation equality (14).

For the second block the preservation equality

$$\overline{x_2} + s_{1,1} = s_{2,1} + s_{2,2} \tag{16}$$

needs to be derived. The variables $s_{2,1}$ and $s_{2,2}$ are defined by the reification constraints

$$\begin{aligned} &(\text{ID: } 4) && \bar{s}_{2,1} + \bar{x}_2 + s_{1,1} \geq 1 && (17a)\\ &(\text{ID: } 5) && 2\bar{s}_{2,2} + \bar{x}_2 + s_{1,1} \geq 2 && (17b)\\ &(\text{ID: } 6) && 2s_{2,1} + x_2 + \bar{s}_{1,1} \geq 2 && (17c)\\ &(\text{ID: } 7) && s_{2,2} + x_2 + \bar{s}_{1,1} \geq 1 && (17d) \end{aligned}$$

These constraints are again introduced by redundance-based strengthening in the proof file.

```
red +1 ~s21 +1 ~x2 +1  s11 >= 1 ; s21 -> 0
red +2 ~s22 +1 ~x2 +1  s11 >= 2 ; s22 -> 0
red +2  s21 +1  x2 +1 ~s11 >= 2 ; s21 -> 1
red +1  s22 +1  x2 +1 ~s11 >= 1 ; s22 -> 1
```

This time some additional steps are required to derive the preservation equality. Adding (17a) and (17b) together yields $\bar{s}_{2,1} + 2\bar{s}_{2,2} + 2\bar{x}_2 + 2s_{1,1} \geq 3$ and dividing by 2 results in

$$(\text{ID: } 8) \qquad \bar{s}_{2,1} + \bar{s}_{2,2} + \bar{x}_2 + s_{1,1} \geq 2\,. \tag{18}$$

Adding (17c) and (17d) yields $2s_{2,1} + s_{2,2} + 2x_2 + 2\bar{s}_{1,1} \geq 3$ and dividing by 2 results in

$$(\text{ID: } 9) \qquad s_{2,1} + s_{2,2} + x_2 + \bar{s}_{1,1} \geq 2\,. \tag{19}$$

These cutting planes derivations are written to the proof log in reverse polish notation using the identifiers for the constraints. The first line derives (18) and the second line derives (19).

```
pol 4 5 + 2 d
pol 6 7 + 2 d
```

The constraints (18) and (19) together represent the desired preservation equality (16).

The next step is to sum the preservation equalities together with the input constraint (13). The sum of the constraints (15b) and (19) is $\overline{x_1} + x_2 + s_{2,1} + s_{2,2} \geq 2$ and adding (13) yields

$$\text{(ID: 10)} \qquad s_{2,1} + s_{2,2} \geq 2\,. \tag{20}$$

This cutting planes derivation is written to the proof log as the following line.

```
pol 3 9 + 1 +
```

The next step is to derive the clauses

$$\text{(ID: 11)} \qquad \bar{x}_1 + s_{1,1} \geq 1 \tag{21a}$$
$$\text{(ID: 12)} \qquad x_1 + \bar{s}_{1,1} \geq 1 \tag{21b}$$
$$\text{(ID: 13)} \qquad x_2 + s_{2,1} \geq 1 \tag{21c}$$
$$\text{(ID: 14)} \qquad \bar{s}_{1,1} + s_{2,1} \geq 1 \tag{21d}$$
$$\text{(ID: 15)} \qquad \bar{x}_2 + s_{1,1} + \bar{s}_{2,1} \geq 1 \tag{21e}$$
$$\text{(ID: 16)} \qquad x_2 + \bar{s}_{1,1} + s_{2,2} \geq 1 \tag{21f}$$
$$\text{(ID: 17)} \qquad \bar{x}_2 + \bar{s}_{2,2} \geq 1 \tag{21g}$$
$$\text{(ID: 18)} \qquad s_{1,1} + \bar{s}_{2,2} \geq 1 \tag{21h}$$

that encode the sequential counter as introduced in (7). The clauses in (21) can be derived by RUP, which is specified in the proof log by the following lines.

```
rup +1 ~x1           +1  s11 >= 1 ;
rup +1  x1           +1 ~s11 >= 1 ;
rup +1  x2           +1  s21 >= 1 ;
rup +1 ~s11          +1  s21 >= 1 ;
rup +1 ~x2  +1  s11 +1 ~s21 >= 1 ;
rup +1  x2  +1 ~s11 +1  s22 >= 1 ;
rup +1 ~x2           +1 ~s22 >= 1 ;
rup +1  s11          +1 ~s22 >= 1 ;
```

To see that these clauses follow by reverse unit propagation, we detail the RUP step for (21c) and the RUP step for the other clauses is similar. The negation of (21c) is $\bar{x}_2 + \bar{s}_{2,1} \geq 2$, which propagates x_2 and $s_{2,1}$ to false. This falsifies (17c), hence (21c) is implied.

The last step is to enforce the comparison with the degree of the constraint. As $x_1 + \bar{x}_2 \geq 2$ is satisfied if $s_{2,2}$ is true, the constraint

$$\text{(ID: 19)} \qquad s_{2,2} \geq 1 \tag{22}$$

has to be derived to enforce the comparison. This can be done using RUP, which is also written to the proof log.

```
rup +1 s22 >= 1 ;
```

This concludes our example.

To obtain the encoding with k-simplification, the most naive approach would be to simply omit the clauses enforcing correct values for the variables $s_{i,j}$ that are not used. However, this could incur a significant overhead in the proof logging when k is small, as we would always introduce $\Theta(n^2)$ intermediate variables instead of the $\Theta(kn)$ variables actually used in the final encoding. To avoid this overhead, we can introduce "overflow variables" $s_{i,k+2}$ that do not encode that the first i bits sum to $k + 2$ but instead ensure that the equality

$$\ell_i + \textstyle\sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \tag{23}$$

holds. To maintain the equality of sums over incoming and outgoing edges in our graph representation, we label the edge to the next block by $\sum_{j=1}^{k+1} s_{i,j}$ instead of $\sum_{j=1}^{i} s_{i,j}$, and introduce an additional edge going directly to the sink with the label $s_{i,k+2}$ (see Figure 3b). Note that without the additional variable $s_{i,k+2}$ we could not guarantee equality, as we would have $k + 2$ literals on the left-hand side and only $k + 1$ variables on the right-hand side.

Example 2. To apply k-simplification for $k = 1$ to Figure 3a, the output from block 3 to block 4 should only contain the sum of the two variables $s_{3,1} + s_{3,2}$. To preserve equality of the sums of inputs and outputs, we add an edge from block 3 to the sink labelled $s_{3,3}$ as in Figure 3b.

When using k-simplification, we can derive an analogue of (8) by a telescoping sum of all preservation equalities (23) yielding $\sum_{i=1}^{n} \left(\ell_i + \sum_{j=1}^{k+1} s_{i-1,j}\right) = \sum_{i=1}^{n} \left(\sum_{j=1}^{k+2} s_{i,j}\right)$, which simplifies to $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}$.

4 A General Framework for Certifying CNF Translations

As discussed in the introduction, there is a rich selection of encodings of pseudo-Boolean constraints in CNF. In this section, we develop a unified framework to provide proof logging for a wide range of different translations. Our approach is to represent encodings as directed graphs with preservation equalities between the incoming and outgoing edges of each node, as in our example in Figure 3, so that all clauses in the encoding can be obtained by reverse unit propagation from (telescoping sums over) these equalities. In this way, the whole proof logging task is reduced to considering a few generic ways of deriving preservation equalities. Let us start with a formal definition of the graph representation.

We will describe how the proof logging works by first introducing concrete methods that provide proof logging for different low-level steps, and then showing how these methods can be composed to certify correctness of translations from pseudo-Boolean constraints to CNF. Recall that every constraint is assigned a unique identifier. A cutting planes derivation is specified by $\mathsf{add}\,(C, D)$ to add C and D together, $\mathsf{mult}\,(C, k)$ to multiply C by k and $\mathsf{div}\,(C, k)$ to divide C by k and round up. E.g., given the previously derived constraints C and D, calling $\mathsf{add}\,(\mathsf{div}\,(C, 2)\,, \mathsf{mult}\,(D, 3))$ divides C by 2 (and rounds up), multiplies D by 3, adds the two constraints obtained in this way together, returns the resulting constraint, and writes the corresponding derivations to the proof file in reverse polish notation and using the identifiers for the constraints. A reverse unit propagation constraint C can be added using $\mathsf{rup}\,(C)$. The syntax we use for deriving a constraint by reification is $\mathsf{red}\,(z \Rightarrow C, \{z \rightarrow 0\})$ and $\mathsf{red}\,(z \Leftarrow C, \{z \rightarrow 1\})$ (where this somewhat cryptic notation is due to the fact that reification is a special case of the redundance rule in [GN21]). We use ▹ to denote comments in the pseudocode.

Definition 1 (Arithmetic Graph)**.** Let a_i, c_i be integers, ℓ_i Boolean literals, and o_i Boolean variables. An *arithmetic graph* with input $\sum_i a_i \ell_i$ and output $\sum_i c_i o_i$ is a directed multi-graph $G = (V, E)$ that satisfies the following conditions:

1. Every edge $e \in E$ has a label of the form $\sum_i b_i^e y_i^e$ for each edge $e \in E$, where b_i^e are integers and y_i^e Boolean variables.

2. There is a unique source node s that has only outgoing edges, and these edges are labelled by input literals ℓ_i in such a way that $\sum_i a_i \ell_i = \sum_{(s,v)=e \in E} \sum_i b_i^e y_i^e$.

3. There is a unique sink t that has only incoming edges, and these edges are labelled by output variables o_i in such a way that $\sum_i c_i o_i = \sum_{(v,t)=e \in E} \sum_i b_i^e y_i^e$.

4. For all other nodes v, which we refer to as *inner nodes*, the preservation equality
$$\sum_{(u,v)=e \in E} \sum_i b_i^e y_i^e = \sum_{(v,w)=e \in E} \sum_i b_i^e y_i^e \tag{24}$$
has to hold. This is saying that the sum of incoming edges equals the sum of outgoing edges, which can be derived using cutting planes with reification over the variables on outgoing edges from v.

The arithmetic graph does not necessarily have to be acyclic, but an acyclic graph simplifies the arguments for correctness of the generated proof.

The rest of this section will be devoted to discussing how preservation equalities (24) can be derived for different types of pseudo-Boolean expressions. Before doing so, let us just note for the record that if we have an arithmetic graph for an encoding of a pseudo-Boolean constraint, then by a telescoping argument as in Section 3 we can derive that the same constraint applies to the output of the graph.

Proposition 1. *Given an arithmetic graph with input $\sum_i a_i \ell_i$ and output $\sum_i c_i o_i$ and a PB constraint $\sum_i a_i \ell_i \bowtie k$ for $\bowtie \in \{\geq, \leq, =\}$, we can derive $\sum_i c_i o_i \bowtie k$ using cutting planes.*

Algorithm 4: General algorithm for translating PB constraints to CNF with proof logging.

```
1 translate_and_certify($C, f, G, F$)
    ▹ input: pseudo-Boolean constraint $C$ of the form $\sum_{i=1}^{n} a_i\ell_i \bowtie k$, with
      $\bowtie \in \{\geq, \leq, =\}$
    ▹ input: arithmetic graph $G = (V, E)$ with input $\sum_i a_i\ell_i$ and output
      $\sum_i c_i o_i$
    ▹ input: function $f$ that takes a node and derives its preservation
      equality
    ▹ input: set of clauses $F$ with CNF encoding to be derived
2   sum constraints $f(v)$ for $v \in V$ in topological order to obtain
      $\sum_i a_i\ell_i = \sum_i c_i o_i$;
3   combine $\sum_i a_i\ell_i = \sum_i c_i o_i$ and $C$ to obtain $\sum_i c_i o_i \bowtie k$;
4   derive each clause in the CNF encoding $F$ with reverse unit
      propagation (RUP);
```

Proof. By item 4 in Definition 1, we can derive preservation equalities (24) for all inner nodes in the graph. By summing the preservation equalities for all inner nodes together (i.e., adding up separately all greater-than-or-equal constraints and all less-than-or-equal constraints, as explained in Section 2), we obtain $\sum_i a_i\ell_i = \sum_i c_i o_i$, and combining this with $\sum_i a_i\ell_i \bowtie k$ yields $\sum_i c_i o_i \bowtie k$ as desired. □

Once the bound on the input literals is translated to a bound on the output variables, all clauses of the CNF encoding will follow by reverse unit propagation. This results in the general proof logging method shown in Algorithm 4. Note that the nodes of the graph should be traversed in topological order when deriving the preservation equalities—this is so that the variables used in the reification steps are all fresh.

Let us now discuss three different ways of representing values of natural numbers that are used in preservation equality for inner nodes. Perhaps the most straightforward way to encode a number j with domain $A = \{0, 1, \ldots, m\} \subseteq \mathbb{N}_0$ with Boolean variables is to write j in unary with variables z_i so that $j = \sum_{i\in[m]} z_i$. In such an encoding we can also require, using constraints $z_i \geq z_{i+1}$, that the variables z_i are ordered so that z_i is true if and only if $j \geq i$. This means that listing the variables in reverse order $z_m, z_{m-1}, \ldots, z_1$ yields the number j written in unary (after a prefix of zeros). This is known as the *order encoding*, and this type of representation is used in the sequential counter [Sin05] and totalizer [BB03] encodings. We can certify the correctness of this encoding as stated in the next proposition.

Algorithm 5: Deriving a unary sum over fresh variables z_i.

```
derive_unary_sum($C'$)
    ▷ input: $C'$ of the form $\sum_{i=1}^{n}\ell_i = \sum_{i=1}^{n} z_i$ and describing the constraint to be derived
    ▷ the $z_i$ variables need to be fresh, the left-hand side is the sum to be encoded
    for $j = 1, \ldots, k$ do
        ▷ Step 5.1: introduce variables
        $D_j^{\Rightarrow}, D_j^{\Leftarrow} \leftarrow \text{reify}(z_j \Leftrightarrow \sum_{i=1}^{n} 1 \cdot \ell_i \geq j)$;
    ▷ Step 5.2: derive $\sum_{i=1}^{n}\ell_i \geq \sum_{i=1}^{n} z_i$
    $C^{\Rightarrow} \leftarrow$ derive_sum($D_1^{\Rightarrow}, D_2^{\Rightarrow}, \ldots, D_n^{\Rightarrow}$);
    ▷ Step 5.3: derive $\sum_{i=1}^{n}\ell_i \leq \sum_{i=1}^{n} z_i$
    $C^{\Leftarrow} \leftarrow$ derive_sum($D_n^{\Leftarrow}, D_{n-1}^{\Leftarrow}, \ldots, D_1^{\Leftarrow}$);
    for $i = 1, \ldots, k-1$ do
        ▷ Step 5.4: derive $z_i \geq z_{i+1}$, $i \in [n-1]$
        derive_ordering($D_i^{\Leftarrow}, D_{i+1}^{\Rightarrow}$);
    return $C^{\Rightarrow}, C^{\Leftarrow}$;
```

Algorithm 6: Reify $\sum_{i=1}^{n} a_i\ell_i \geq j$ using the fresh variable z_j.

```
reify($z_j \Leftrightarrow \sum_{i=1}^{n} a_i\ell_i \geq j$)
    ▷ $z_j \Rightarrow \sum_{i=1}^{n} a_i\ell_i \geq j$
    $C^{\Rightarrow} \leftarrow \text{red}\left(\sum_{i=1}^{n} a_i\ell_i + j\bar{z}_j \geq j, \{z_j \rightarrow 0\}\right)$;
    ▷ $z_j \Leftarrow \sum_{i=1}^{n} a_i\ell_i \geq j$
    $C^{\Leftarrow} \leftarrow \text{red}\left(\sum_{i=1}^{n} a_i\bar{\ell}_i + (\sum_{i=1}^{n} a_i - j + 1)z_j \geq \sum_{i=1}^{n} a_i - j + 1, \{z_j \rightarrow 1\}\right)$;
    return $C^{\Rightarrow}, C^{\Leftarrow}$;
```

Proposition 2 (Unary Sum). *For literals ℓ_i and fresh variables z_i, $i \in [n]$, the constraints*

$$\sum_{i=1}^{n}\ell_i \geq \sum_{i=1}^{n} z_i \tag{25a}$$

$$\sum_{i=1}^{n}\ell_i \leq \sum_{i=1}^{n} z_i \tag{25b}$$

$$z_i \geq z_{i+1} \qquad i \in [n-1] \tag{25c}$$

can be derived in O(n) *steps in cutting planes with reification. Thus, the variable z_i is defined to be true if and only if at least i literals are true.*

Proof. The unary sum constraints in (25) can be derived using Algorithm 5. We will show the correctness of Algorithm 5 first and then that the derivation following this algorithm requires O(n) steps in cutting planes with reification.

Algorithm 7: Derive sum of reification variables.

```
1 derive_sum($D_1, \dots, D_n$)
      ▹ input: $D_j$ is of the form $\sum_{i=1}^{n} \ell_i + j\overline{z}_j \geq j$
2     $C \leftarrow 0 \geq 0$;
3     for $j$ from 1 to $n$ do
4         $C \leftarrow \mathsf{div}\left(\mathsf{add}\left(\mathsf{mult}(C, j-1), D_j\right), j\right)$;
          ▹ Invariant: $C : \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$
5     return $C$;
```

Algorithm 8: Deriving an ordering constraint $z_A \geq z_B$ from the reification constraints.

```
1 derive_ordering($C, D$)
      ▹ input: $C$ is of the form $z_A \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq A$
      ▹ input: $D$ is of the form $z_B \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq B$
2     $divisor \leftarrow \sum_{i=1}^{n} a_i$;
      ▹ derive $z_A \geq z_B$ if $A < B$
3     $\mathsf{div}(\mathsf{add}(C, D), divisor)$;
```

Algorithm 5 is split into four major steps. Step 5.1 is to introduce the fresh variables z_j as reifications of the constraints $\sum_{i=1}^{n} \ell_i \geq j$, which is shown in Algorithm 6 for the more general case of arbitrary positive coefficients.

In Step 5.2 the lower bound (25a) is derived using Algorithm 7 maintaining the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$ after each iteration. For the base case $j = 1$, the invariant is equivalent to the reification constraint $z_1 \Rightarrow \sum_{i=1}^{n} \ell_i \geq 1$, which in normalized form is $\sum_{i=1}^{n} \ell_i + \overline{z}_1 \geq 1$ and hence this case is covered. For the inductive step, to go from $j-1$ to j we multiply the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j-1} \overline{z}_j \geq j-1$ by $j-1$ and add the reification constraint $z_j \Rightarrow \sum_{i=1}^{n} \ell_i \geq j$, which is $\sum_{i=1}^{n} \ell_i + j\overline{z}_j \geq j$ in normalized form, to get $j\sum_{i=1}^{n} \ell_i + (j-1)\sum_{i=1}^{j-1} \overline{z}_i + j\overline{z}_j \geq j^2 - j + 1$. Division by j and rounding up yields $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i + \overline{z}_j \geq j$, i.e., the invariant for j. For $j = n$ the invariant is the normalized form of (25a).

In Step 5.3 the upper bound (25b) is again derived using Algorithm 7, except that the constraints are processed in reverse order (just as in Example 1 on page 69).

In Step 5.4 the ordering constraint is derived using Algorithm 8, using the reification constraints. Algorithm 8 handles the general case of deriving the ordering constraint $z_j \geq z_{j+1}$ from any reification constraints $z_{j+1} \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j+1$ and $z_j \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq j$. For the sequential counter encoding the coefficients a_i are all 1. In normalized form these two constraints are $(j+1)\overline{z}_{j+1} + \sum_{i=1}^{n} a_i \ell_i \geq j+1$ and $(m-j+1)z_j + \sum_{i=1}^{n} a_i \overline{\ell}_i \geq m-j+1$, where $m = \sum_{i=1}^{n} a_i$. Adding both constraints together yields $(m - j + 1)z_j + (j + 1)\overline{z}_{j+1} \geq 2$ and we get the desired ordering constraint after division by a large enough number, such as m.

Now that the correctness of Algorithm 5 is established, all that remains is to verify that this algorithm uses $\mathrm{O}(n)$ steps in cutting planes with reification. Step 5.1 uses n reification steps, Step 5.2 and 5.3 uses $3(n - 1)$ cutting planes steps each and Step 5.4 uses $2(n - 1)$ cutting planes steps in the worst case. Thus, in total $\mathrm{O}(n)$ steps in cutting planes with reification are used. □

A concrete illustration of how these derivations can be done was given in Example 1 (with ℓ_3, $s_{2,1}$, and $s_{2,2}$ playing the roles of the literals ℓ_i and $s_{3,j}$, $j \in [3]$, being the fresh variables).

When encoding the value of a number j that can only take a small number of values in a large range, it is wasteful to introduce variables for all values in the range. For example, if $j \in \{0, 50, 75\}$, then the first 50 variables in a full unary representation are either all true or all false, but will never take different values. In such cases we can instead use what we will refer to as a *sparse unary encoding*, where in our example $j \in \{0, 50, 75\}$ would be represented as $50 \cdot z_{50} + 25 \cdot z_{75}$, where we enforce $z_{50} \geq z_{75}$. More formally, for a (finite) domain $A \subseteq \mathbb{N}_0$ and variables $\vec{z} = \{z_i \mid i \in A \cup \{\infty\}\}$ we define

$$\mathsf{sparse}(\vec{z}, A) \doteq \textstyle\sum_{i \in A \setminus \{0\}} (i - \mathsf{pred}(i, A)) \cdot z_i \,, \tag{26a}$$

where $\mathsf{pred}(i, A) = \max\{j \in A \cup \{0\} \mid j < i\}$, and we also use constraints

$$z_i \geq z_{\mathsf{succ}(i,A)} \qquad\qquad i \in A \setminus \{\max(A)\} \tag{26b}$$

to enforce that the variables z_i are ordered, where $\mathsf{succ}(i, A) = \min\{j \in A \cup \{\infty\} \mid j > i\}$ is the successor of i in A. This representation is used in the sequential weight counter [HMS12] and generalized totalizer [JMM15] encodings, and we can certify correctness for it as stated next.

Proposition 3 (Sparse Unary Sum). *Let $A, B \subseteq \mathbb{N}_0$ be given with sparse encodings* $\mathsf{sparse}(\vec{x}, A)$ *and* $\mathsf{sparse}(\vec{y}, B)$ *as in* (26a)–(26b). *Then for $E = \{i + j \mid i \in A, j \in B\}$ and fresh variables $\vec{z}$ we can derive*

$$\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) = \mathsf{sparse}(\vec{z}, E) \tag{27a}$$

$$z_i \geq z_{\mathsf{succ}(i,E)} \qquad\qquad i \in E \setminus \{\max(E)\} \tag{27b}$$

in cutting planes with reification using $\mathrm{O}(|A| \cdot |B|)$ *steps.*

Proof. The proposition is proven by presenting and analyzing Algorithm 9, which given two numbers in sparse unary representation derives their sum. Just as for the unary sum, we start in Step 9.1 by introducing the required fresh variables via reification. However, we only need to introduce the variables with index in E. If k-simplification is used, then also variables with index bigger than k need to be introduced, as without them equality cannot be derived. The ordering constraints can be derived as before using Algorithm 8.

Algorithm 9: Deriving a sparse unary sum over fresh variables $\vec{z}$.

```
derive_sparse_unary_sum($C'$)
    ▷ input: $C'$ of the form $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) = \mathsf{sparse}(\vec{z}, E)$ and
      describing the constraint to be derived such that $A, B \subseteq \mathbb{N}$,
      $E = \{i + j | i \in A, j \in B\}$
    ▷ Step 9.1: Introduce variables as reification and derive ordering
    for $j \in E \setminus \{0\}$ do
        $D_j^{\Rightarrow}, D_j^{\Leftarrow} \leftarrow \mathrm{reify}(z_j \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq j)$;
    for $i \in E \setminus \{0, \max(E)\}$ do
        derive_ordering ( $D_i^{\Leftarrow}, D_{\mathsf{succ}(i,E)}^{\Rightarrow}$ ) ;        ▷ derive $z_i \geq z_{\mathsf{succ}(i,E)}$
    ▷ Step 9.2: : reify constraint to be derived
    $C^{\Rightarrow}, \_ \leftarrow \mathrm{reify}(z_{geq} \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq \mathsf{sparse}(\vec{z}, E))$;
    $C^{\Leftarrow}, \_ \leftarrow \mathrm{reify}(z_{leq} \Leftrightarrow \mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \leq \mathsf{sparse}(\vec{z}, E))$;
    $\mathrm{reify}(z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2)$;
    ▷ Step 9.3: derive that $z_{eq} \geq 1$
    try_all_values($\mathsf{sparse}(\vec{x}, A), \mathsf{sparse}(\vec{y}, B), z_{eq}$);
    ▷ Step 9.4: derive constraint to be derived from its reification
    $M \leftarrow \max(E)$ ; ▷ choose $M$ equal to coefficient of reification variables
    $D \leftarrow \mathsf{rup}\left(z_{geq} \geq 1\right)$;
    $C^{\Rightarrow} \leftarrow \mathsf{add}\,(C^{\Rightarrow}, \mathsf{mult}\,(D, M))$;
    $D \leftarrow \mathsf{rup}\left(z_{leq} \geq 1\right)$;
    $C^{\Leftarrow} \leftarrow \mathsf{add}\,(C^{\Leftarrow}, \mathsf{mult}\,(D, M))$;
    return $C^{\Rightarrow}, C^{\Leftarrow}$;
```

In Step 9.2 we introduce a variable z_{eq} which is true if and only if the equality to be derived is true. Since an equality is actually two inequalities, we need to introduce separate variables z_{geq}, z_{leq} for each inequality and then combine them into z_{eq}.

In Step 9.3 we derive $z_{eq} \geq 1$ by checking all combinations of values in A and B, which requires $O(|A| \cdot |B|)$ steps.

In Step 9.4 we use that $z_{eq} \geq 1$ and hence $z_{geq} = z_{leq} = 1$, which allows us to derive $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \geq \mathsf{sparse}(\vec{z}, E)$ and $\mathsf{sparse}(\vec{x}, A) + \mathsf{sparse}(\vec{y}, B) \leq \mathsf{sparse}(\vec{z}, E)$ by removing z_{geq} and z_{leq} from the constraints introduced in Step 9.2.

Since Step 9.1 and 9.3 require $O(|A| \cdot |B|)$ steps each and the number of steps for Step 9.2 and 9.4 is in $O(1)$, the total number of cutting planes with reification steps is $O(|A| \cdot |B|)$. Asymptotically, this is the same number of steps required to compute which elements are in E, so this is still linear in the time needed to construct the encoding. □

As in the case of the unary sum in Proposition 2, adding the constraints (27a)–(27b) maintains equisatisfiability, because the fresh variables $\vec{z}$ are free to

Algorithm 10: Given a reified sparse unary sum, derive that the reification variable is true.

```
▷ helper function:
fix(sparse($\vec{x}, A$), $a$)
    return $\overline{x}_a + x_{\mathsf{succ}(a,A)}$ ;                ▷ replace $x_0$ by 1 and $x_\infty$ by 0
▷ main function:
try_all_values(sparse($\vec{x}, A$), sparse($\vec{y}, B$), $z_{eq}$)
    $C_{outer} \leftarrow$ rup $(0 \geq 0)$;
    for $i \in A$ do
        $C_{inner} \leftarrow$ rup $(0 \geq 0)$;
        for $j \in B$ do
            ▷ $a$ (respectively $b$) is the value encoded by sparse($\vec{x}, A$)
              (sparse($\vec{y}, B$))
            ▷ encode that $(a = i \wedge b = j) \Rightarrow z_{eq}$
            $D \leftarrow$ rup (fix(sparse($\vec{x}, A$), $i$)
                    + fix(sparse($\vec{y}, B$), $j$) + $z_{eq} \geq 1$) ;
            $C_{inner} \leftarrow$ add $(C_{inner}, D)$;
        $C_{outer} \leftarrow$ add $(C_{outer}$, div $(C_{inner}, |B|))$;
    $C_{outer} \leftarrow$ div $(C_{outer}, |A|)$;
    return $C_{outer}$ ;                                    ▷ $C_{outer}$ is now $z_{eq} \geq 1$
```

take values so that the constraints are satisfied. The general idea is again to introduce $\vec{z}$ via reification, but the rest of the proof of Proposition 3 gets a bit more complicated—we have to perform a brute-force search on the possible combinations of values for A and B, showing that the equality holds in all cases, and provide a proof log for the correctness of this backtracking search.

To illustrate how the derivation in Algorithm 9 works, let us consider an example.

Example 3. Let the set of possible values for the left child node be $A = \{0, 2\}$ and the corresponding set for the right child node be $B = \{0, 2, 4\}$. Hence, the set of possible output values is $E = \{0, 2, 4, 6\}$. Step 9.1 derives the reified constraints

$$z_2 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 2 \tag{28a}$$

$$z_4 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 4 \tag{28b}$$

$$z_6 \Leftrightarrow 2x_2 + 2y_2 + 2y_4 \geq 6 \tag{28c}$$

and the ordering constraints $z_2 \geq z_4$ and $z_4 \geq z_6$.

Then Step 9.2 uses reification to derive the constraints

$$6\overline{z}_{geq} + 2x_2 + 2y_2 + 2y_4 + 2\overline{z}_2 + 2\overline{z}_4 + 2\overline{z}_6 \geq 6 \tag{29a}$$

$$6\overline{z}_{leq} + 2\overline{x}_2 + 2\overline{y}_2 + 2\overline{y}_4 + 2z_2 + 2z_4 + 2z_6 \geq 6 \tag{29b}$$

$$z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2\,. \tag{29c}$$

Then Step 9.3 derives $z_{eq} \geq 1$ using Algorithm 10 by checking all combinations of values in A and B. After the first iteration of the outer loop in Algorithm 10 the clauses

$$x_2 + y_2 + z_{eq} \geq 1 \tag{30a}$$
$$x_2 + \overline{y}_2 + y_4 + z_{eq} \geq 1 \tag{30b}$$
$$x_2 + \overline{y}_4 + z_{eq} \geq 1 \tag{30c}$$

have been derived. Deriving (30a) by RUP sets $x_2 = y_2 = z_{eq} = 0$. This causes the ordering constraints to propagate all variables in $\vec{x}$ and $\vec{y}$. As all $\vec{x}$ and $\vec{y}$ variables are set, the reification constraints introduced in Step 9.1 will cause all $\vec{z}$ variables to propagate. As the constraints reified in Step 9.2 are satisfied, $z_{geq} = z_{leq} = 1$ is propagated and hence z_{eq} should be 1. However, RUP already set z_{eq} to 0, which is a contradiction showing that (30a) can be derived. Deriving the other clauses works analogously. Adding all clauses in (30) together results in $3x_2 + 3z_{eq} \geq 1$, which is divided by 3 to obtain

$$x_2 + z_{eq} \geq 1\,. \tag{31}$$

Analogously, in the second iteration we derive the constraints

$$\overline{x}_2 + y_2 + z_{eq} \geq 1 \tag{32a}$$
$$\overline{x}_2 + \overline{y}_2 + y_4 + z_{eq} \geq 1 \tag{32b}$$
$$\overline{x}_2 + \overline{y}_4 + z_{eq} \geq 1 \tag{32c}$$

using RUP and then

$$\overline{x}_2 + z_{eq} \geq 1 \tag{33}$$

by adding all the constraints in (32) together and dividing the result by 3. Adding the constraints (31) and (33) together yields $2z_{eq} \geq 1$ and dividing by 2 results in $z_{eq} \geq 1$.

Step 9.4 first computes the coefficient of z_{geq} and z_{leq}, which is $M = 6$. Then the constraints $z_{geq} \geq 1$ and $z_{leq} \geq 1$ are derived using RUP by setting either $z_{geq} = 0$ or $z_{leq} = 0$. Then $z_{eq} \geq 1$ propagates $z_{eq} = 1$. However, (29c) propagates $z_{eq} = 0$, which is a contradiction. Then $z_{geq} \geq 1$ and $z_{leq} \geq 1$ are multiplied by 6 and added to (29a) and (29b), respectively. This yields constraints

$$2x_2 + 2y_2 + 2y_4 + 2\overline{z}_2 + 2\overline{z}_4 + 2\overline{z}_6 \geq 6 \tag{34a}$$
$$2\overline{x}_2 + 2\overline{y}_2 + 2\overline{y}_4 + 2z_2 + 2z_4 + 2z_6 \geq 6\,, \tag{34b}$$

which together represent the preservation equality for the sparse unary sum.

If we perform sums repeatedly as in Proposition 3, then the size of the domain can double in every step in the worst case, leading to an exponential explosion (this happens, for instance, if all values in the domains are distinct powers of 2).

Algorithm 11: Proof logging for the encoding of a single full adder.

```
full_adder($x,y,z$)
    $D^{\Rightarrow}_{carry}, D^{\Leftarrow}_{carry} \leftarrow$ reify($c \Leftrightarrow x + y + z \geq 2$);
    $D^{\Rightarrow}_{sum}, D^{\Leftarrow}_{sum} \leftarrow$ reify($s \Leftrightarrow x + y + z + 2\bar{c} \geq 3$);
    $D^{\Rightarrow} \leftarrow \mathsf{div}\left(\mathsf{add}\left(\mathsf{mult}\left(D^{\Rightarrow}_{carry}, 2\right), D^{\Rightarrow}_{sum}\right), 3\right)$;
    $D^{\Leftarrow} \leftarrow \mathsf{div}\left(\mathsf{add}\left(\mathsf{mult}\left(D^{\Leftarrow}_{carry}, 2\right), D^{\Leftarrow}_{sum}\right), 3\right)$;
    ▷ $D$ is the preservation equality of the full adder
    return $D^{\Rightarrow}, D^{\Leftarrow}, c, s$;
```

The third encoding we consider addresses this worst-case scenario by using a *binary encoding* $j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i \cdot z_i$. To compute the binary representation, it is sufficient—as we will discuss next in Section 5—to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit as described in [ES06].

Proposition 4. *For literals ℓ_1, ℓ_2, ℓ_3 and fresh variables c, s, we can derive the equality*

$$\ell_1 + \ell_2 + \ell_3 = 2c + s \tag{35}$$

in cutting planes with reification using O(1) *steps.*

Proof. Algorithm 11 can be used to derive the constraints that represent the preservation equality (35) for a single binary full adder.

Algorithm 11 first derives

$$c \Leftrightarrow \ell_1 + \ell_2 + \ell_3 \geq 2 \tag{36a}$$

$$s \Leftrightarrow \ell_1 + \ell_2 + \ell_3 + 2\bar{c} \geq 3 \tag{36b}$$

using reification, since c and s are fresh variables, and then multiplies (36a) by 2, add (36b), and divides the result by 3. To show how this works for the $\Rightarrow$-direction of the reification, 2 times (36a) is $4\bar{c} + 2\ell_1 + 2\ell_2 + 2\ell_3 \geq 4$, adding $3\bar{s} + \ell_1 + \ell_2 + \ell_3 + 2\bar{c} \geq 3$ as in (36b) yields $6\bar{c} + 3\bar{s} + 3\ell_1 + 3\ell_2 + 3\ell_3 \geq 7$, and dividing by 3 gives us $2\bar{c} + \bar{s} + \ell_1 + \ell_2 + \ell_3 \geq 3$ as desired. The other direction is equivalent. We refer the reader to [GN21] for more details.

This algorithm uses 2 reification steps and 6 cutting planes steps. Thus, the number of cutting planes with reification steps is in O(1). □

Again, it should be clear that this maintains equisatisfiability, since the carry-out bit c and sum bit s can be set appropriately.

Algorithm 12: Construction of adder network [ES06]. Procedure full_adder adds full adder to network.

```
adder_network($b$)
    ▷ input: vector of buckets $b$
    for $i$ from 0 to $b$.size() do
        while $b_i$.size() $\geq 2$ do
            if $b_i$.size() $= 2$ then
                $(x, y) \leftarrow b_i$.dequeue();
                $(c, s) \leftarrow$ full_adder$(x, y, 0)$;
            else
                $(x, y, z) \leftarrow b_i$.dequeue();
                $(c, s) \leftarrow$ full_adder$(x, y, z)$;
            $b_i$.enqueue($s$);
            $b_{i+1}$.enqueue($c$);
```

5 Certifying the Binary Adder Network Encoding

Now that this general framework has been introduced, we show how it can be applied to implement proof logging for some specific pseudo-Boolean to CNF encodings. In this section, we will consider the so-called *binary adder encoding* [ES06].

The idea behind the binary adder encoding is to use an adder network to compute the value of $\sum_i a_i \ell_i$ as a binary number $\sum_{i=0}^{bits} 2^i o_i$, where $bits = \lfloor \log_2(\sum_i a_i) \rfloor$ is the required *bit width*, and then compare this to the right-hand side constant in the constraint $\sum_i a_i \ell_i \bowtie k$.

To recapitulate the algorithm for adder network construction in [ES06], let us say that a 2^m*-bit* is a literal representing the numerical value 2^m and that a 2^m*-bucket* is a queue of 2^m-bits. We use $[m]_2$ to denote the binary representation of a natural number m. The algorithm starts by initializing each 2^m-bucket with all literals ℓ_i in $\sum_i a_i \ell_i \bowtie k$ such that the 2^m-bit of $\left[a_i\right]_2$ is 1. Then for m in increasing order we repeat the following procedure: while there are at least 2 elements in the 2^m-bucket, dequeue three bits x, y, z, or set $z = 0$ if there are exactly 2 bits left. Use x, y, and z as input for a new full adder with fresh variables c and s as output (these are just placeholder names), and insert s in the 2^m-bucket and c in the 2^{m+1}-bucket (possibly creating a new bucket). See Algorithm 12 for the pseudocode to generate this encoding.

The arithmetic graph is obtained from the adder network by representing each full adder by a node. Each inner node constructed from a 2^m-bucket has 3 input edges with labels $2^m \cdot x$, $2^m \cdot y$, and $2^m \cdot z$ and 2 output edges with labels $2^m \cdot s$ and $2^{m+1} \cdot c$. An example for the PB expression $5x_1 + 4x_2 + x_3 + x_4 + x_5$ is shown in Figure 4. The preservation equality can be derived using Proposition 4 and multiplying the

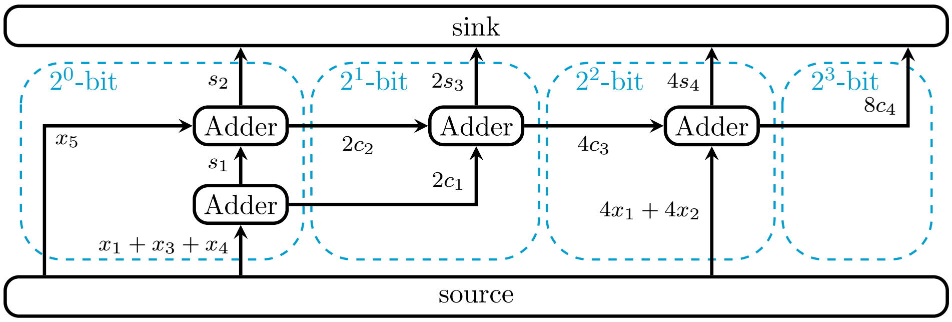


Figure 4: *Layout of arithmetic graph for adder network encoding of* $5x_1 + 4x_2 + x_3 + x_4 + x_5$.

resulting equality $x+y+z = 2c+s$ by 2^m to obtain $2^m \cdot x + 2^m \cdot y + 2^m \cdot z = 2^{m+1} \cdot c + 2^m \cdot s$. When the construction algorithm ends, each 2^m-bucket has at most one 2^m-bit left, and we connect the corresponding edges to the sink, resulting in an output of the form $\sum_{i=0}^{bits} 2^i \cdot o_i$. If the 2^i-bucket is empty, o_i is fixed to 0.

Each full adder of the network is encoded to CNF using clauses

$$\begin{array}{llll} & \bar{x} + \bar{y} + \bar{z} + s \geq 1 & & x + y + z + \bar{s} \geq 1 \\ \bar{y} + \bar{z} + c \geq 1 & \bar{x} + y + z + s \geq 1 & y + z + \bar{c} \geq 1 & x + \bar{y} + \bar{z} + \bar{s} \geq 1 \\ \bar{x} + \bar{z} + c \geq 1 & x + \bar{y} + z + s \geq 1 & x + z + \bar{c} \geq 1 & \bar{x} + y + \bar{z} + \bar{s} \geq 1 \\ \bar{x} + \bar{y} + c \geq 1 & x + y + \bar{z} + s \geq 1 & x + y + \bar{c} \geq 1 & \bar{x} + \bar{y} + z + \bar{s} \geq 1 \end{array} \tag{37}$$

which are all RUP with respect to the preservation equality $x + y + z = 2c + s$.

To compare the constant k in the PB constraint with the output of the circuit, we encode a bitwise comparison $\vec{x} \geq \vec{y}$ for bit vectors $\vec{x}$ and $\vec{y}$, where $\vec{x} = o_{bits} \cdots o_1 o_0$ and $\vec{y} = [k]_2$ or vice versa, depending on whether we want to encode $\sum_{i=1}^{n} a_i \ell_i \geq k$ or $\sum_{i=1}^{n} a_i \ell_i \leq k$, respectively. The following encoding is standard and can also be found in [ES06]. For $\sum_{i=1}^{n} a_i \ell_i = k$, comparisons for both directions are performed. If the sizes of the two vectors are different, the shorter vector is padded with 0, after which the constraints

$$x_i + \bar{y}_i + \sum_{j=i+1}^{bits} x_j \bar{y}_j + \bar{x}_j y_j \geq 1 \qquad i = 0, 1, \ldots, bits \tag{38}$$

are added to the CNF encoding. Since either $\vec{x}$ or $\vec{y}$ is a vector of constant bits, the constraints (38) are indeed clauses. Basically, the encoding compares the two numbers from the most-significant bit to the least-significant bit. It is only required to check the biggest first index i, where x_i and y_i are different. Then the corresponding clause (38) for index i is only satisfied if $x_i \geq y_i$. The clauses (38) are RUP with respect to the constraint $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$, which we obtain from the arithmetic graph using Proposition 1. To see this, note that asserting (38) to false will set all 2^j-bits for $j > i$ equal but the 2^i-bits to opposite values, which immediately falsifies $\sum_{i=0}^{bits} 2^i \cdot o_i \bowtie k$.

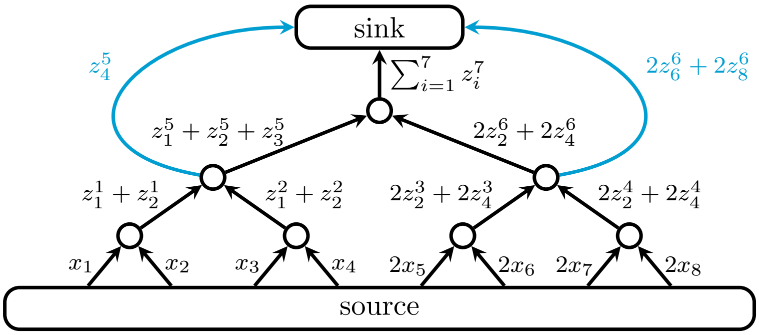


Figure 5: *Layout of the arithmetic graph for the generalized totalizer encoding of* $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$. *Edges introduced for k-simplification are colored cyan.*

6 Certifying the Totalizer and Generalized Totalizer Encodings

To show that the framework developed in Section 4 can be applied to many different pseudo-Boolean to CNF encodings, we detail in this section how our framework can be applied to add certification to the totalizer [BB03] and generalized totalizer [JMM15] encoding.

The totalizer and generalized totalizer encodings accumulate the input in the form of a balanced binary tree. The totalizer encodes cardinality constraints and uses the order encoding to represent values, while the generalized totalizer encodes general pseudo-Boolean constraints and uses a sparse representation. An example of an arithmetic graph for the generalized totalizer encoding is shown in Figure 5. The nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node. To perform k-simplification, the arithmetic graph has additional edges that go directly to the sink node. The formal definition of the arithmetic graph for the (generalized) totalizer encoding is as follows.

Definition 2 (Arithmetic graph for the generalized totalizer encoding)**.** Given a linear sum $\sum_i a_i\ell_i$ over n variables, let G be a binary tree with edges directed towards the root r, leaves s_i for $i \in [n]$ and an additional sink node t with an edge (r, t). The edge (s_i, v) is labelled with $a_i\ell_i$. For an inner node v with two incoming edges labelled $\mathsf{sparse}(\vec{x}, A)$ and $\mathsf{sparse}(\vec{y}, B)$, the outgoing edge is labelled $\mathsf{sparse}(\vec{z}, E)$, where $\vec{z}$ are fresh variables and $E = \{i + j \mid i \in A, j \in B\}$. All s_i are combined into a single source node. For k-simplification we split $\mathsf{sparse}(\vec{z}, E) = \sum_{i \in E} a_i z_i$ into $\sum_{i \leq \mathsf{succ}(k,E)} a_i z_i$ and $\sum_{i > \mathsf{succ}(k,E)} a_i c_i$.

To see that this graph is an arithmetic graph, we only need to check that we can derive the preservation equality for each inner node. We can use Proposition 3 to derive the required preservation equality. Proposition 3 also requires to have ordering constraints on the input literals. However, it is easy to see by an inductive argument that the ordering constraints on the literals can also be derived as we process the nodes in topological order. For the base case, edges from source nodes only contain a single literal, which is vacuously ordered. For inner nodes we get the ordering constraints by applying Proposition 3. If E contains all integers between 0 and $\max(E)$, we can use Proposition 2 to derive the preservation equality, which requires $O(|E|)$ steps instead of $O(|A| \cdot |B|)$ steps and hence reduces overhead.

For each inner node in the graph with incoming edge labels $\mathsf{sparse}(\vec{x}, A)$ and $\mathsf{sparse}(\vec{y}, B)$, the (generalized) totalizer encoding contains the clauses

$$\overline{x}_i + \overline{y}_j + z_{i+j} \geq 1 \qquad i \in A, j \in B \qquad (39a)$$

$$x_{\mathsf{succ}(i,A)} + y_{\mathsf{succ}(j,B)} + \overline{z}_{\mathsf{succ}(i+j,E)} \geq 1 \qquad i \in A, j \in B \qquad (39b)$$

for $\mathsf{succ}(i, A) = \min\{j \mid j \in A \cup \{\infty\}, j > i\}$ and for x_0, y_0 replaced by 1 and $x_\infty, y_\infty, z_\infty$ by 0, with ensuing simplification.

For the proof logging of the CNF encoding we can simply add all clauses using reverse unit propagation. A RUP check of (39a) will assign $x_i = y_j = 1$ and $z_{i+j} = 0$. The ordering constraints on $\vec{x}, \vec{y}$ will propagate variables in $\vec{x}, \vec{y}$ to true so that $\mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$ has a value of at least $i + j$, while the ordering constraints on $\vec{z}$ will propagate variables in $\vec{z}$ to false so that $\mathsf{sparse}(z, E)$ can only take a value strictly less than $i + j$. This will violate the preservation equality $\mathsf{sparse}(z, E) = \mathsf{sparse}(x, A) + \mathsf{sparse}(y, B)$, showing that (39a) is indeed a RUP clause. Deriving the clause (39b) works analogously.

To enforce a pseudo-Boolean constraint $\sum_i a_i \ell_i \bowtie k$, we first derive a bound on the output of the arithmetic graph $\sum_i c_i o_i \bowtie k$, using Proposition 1. Then we can derive unit clauses on the output via reverse unit propagation.

To encode $\sum_i a_i \ell_i \geq k$ or $\sum_i a_i \ell_i \leq k$ the unit clause $z_{\mathsf{succ}(k-1,E)} \geq 1$ or $\overline{z}_{\mathsf{succ}(k,E)} \geq 1$ is added, respectively. This clause is RUP, as the derived sum $\sum_i c_i o_i$ has a value of at most $k - 1$ or at least $k + 1$ and thus the constraint $\sum_i c_i o_i \geq k$ or $\sum_i c_i o_i \leq k$ is falsified, respectively. To encode $\sum_i a_i \ell_i = k$ both unit clauses are added.

7 Experimental Evaluation

To evaluate the proof logging methods developed in this paper, we have implemented certified translations to CNF for the sequential counter [Sin05], adder network [ES06], totalizer [BB03], and generalized totalizer [JMM15] encodings in the tool VeritasPBLib which is publicly available at `https://github.com/forge-lab/VeritasPBLib`. This tool takes a pseudo-Boolean formula in OPB format [RM16] and returns a CNF translation with a proof logging certificate. We have employed the verifier VeriPB[1] [GN21, BGMN22] to check the certificate returned by VeritasPBLib, and have used the SAT solver Kissat[2] [BFFH20], in a lightly modified version outputting *DRAT* proofs in pseudo-Boolean format,[3] to

[1]VeriPB is available at `https://gitlab.com/MIAOresearch/software/VeriPB`.

[2]The original version of Kissat is available at `https://fmv.jku.at/kissat/`.

[3]Our modified version of Kissat with pseudo-Boolean proof logging is available at `https://gitlab.com/MIAOresearch/tools-and-utilities/kissat_fork`.

solve the CNF formula. Finally, we have conjoined the certificates from the CNF translation and the SAT solving and verified the end-to-end pipeline with VeriPB. See [GMNO22] for source code and experimental data.

The experiments were conducted on Amazon EC2 r5.large instances (2 vCPU) with Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz CPUs, 16 GB of memory, and gp2 volumes. We ran one process on each instance with a memory limit of 15 GB and a time limit of 7,200 seconds for verifying the proof with VeriPB, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib and SAT solving with Kissat. We gave additional time for verification, which tends to be slower than solving the problem.

Our evaluation aimed to answer the following questions:

1. Can we use our end-to-end framework to verify the results of SAT-based pseudo-Boolean solving, and how efficient is the verification?

2. How long does the verification of the proof log take when compared to the translation of the pseudo-Boolean formula to CNF?

3. How does a verified SAT-based pseudo-Boolean approach compare against other pseudo-Boolean solvers?

4. Can we use our end-to-end framework to certify the optimal solutions of optimization problems, such as Maximum Satisfiability?

7.1 Benchmarks

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB Evaluation 2016.[4] These instances can be partitioned into formulas with (1) only clauses (279 instances), (2) clauses and cardinality constraints (772 instances) referred to as *Card* in what follows, (3) clauses and general PB constraints (444 instances) called *PB*, and (4) clauses, cardinality and general PB constraints (308 instances) called *Card+PB*. Since this work targets the verification of formulas with non-clausal constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques.

Table 1 shows some properties of the benchmarks used in the experimental results, namely, the average number of constraints, the average number of literals in each constraint, and the average size of coefficients associated with each literal. For each average value (*avg*), we also show the respective standard deviation (*std*) and denote it by $avg \pm std$. This information is shown for both cardinality constraints and PB constraints. Since the benchmark set is composed of instances from multiple domains, there is a large variation of values between instances. For example, the number of cardinality constraints for instances in the *Card* benchmark set ranges from 1 to 2,720, whereas the number of PB constraints for instances in the *PB* benchmark set ranges from 1 to 18,798. In the *Card+PB* benchmark set, we

[4]The benchmarks from the Pseudo-Boolean Evaluation 2016 are available at `http://www.cril.univ-artois.fr/PB16/`.

Table 1: *Properties of the pseudo-Boolean formulas used in the experiments.*

		Card	PB	Card+PB
	#Inst.	772	442	308
Card	Avg. #	107.01±252.57	0.00	1,154.43±5,881.78
	Avg. #Lits	36.45±47.43	0.00	16.96±26.57
	Avg. Coeff. Size	1.00±0.00	0.00	1.00±0.00
PB	Avg. #	0.00	1,020.73±2,294.43	33,379.31±18,3229.66
	Avg. #Lits	0.00	24.95±27.60	105.21±109.99
	Avg. Coeff. Size	0.00	204.93±1,118.74	10.79±50.42

Table 2: *Number of translated, solved and verified instances for each encoding.*

			Translation		Solving			
					#Solved		#Verified	
Category	#Inst	Encoding	#CNF	#Veri	SAT	UNSAT	SAT	UNSAT
Card	772	Sequential	772	772	139	480	133	479
		Totalizer	772	772	139	475	130	474
PB	444	Adder	444	444	179	167	178	165
		GTE	425	414	164	162	150	151
Card+PB	308	Seq+Adder	306	296	134	152	128	151

have an even larger dispersion with instances with 1 to 2,378,901 PB constraints and 1 to 75,582 cardinality constraints.

7.2 End-to-End Solving and Verification

Table 2 shows how VeritasPBLib can be used to generate a CNF formula that can be solved by Kissat and verified by VeriPB. For instances with cardinality constraints (*Card*), we use the sequential counter and totalizer encodings to translate those constraints to CNF. For instances with general PB constraints (*PB*), our translations use the adder network and generalized totalizer (*GTE*) encodings. Finally, for instances with both cardinality and general PB constraints (*Card+PB*), we use the sequential counter encoding for cardinality constraints and the adder network encoding for PB constraints, henceforth denoted by *Seq+Adder*. Even though other combinations of cardinality and PB encodings could be explored, the goal of this work is not to find the best-performing encoding but to show that we can verify the final result for a variety of encodings.

The column *#CNF* shows for how many instances VeritasPBLib successfully generated the CNF translation, which is almost all. The exceptions are 19 instances using *GTE* and 2 instances using the *Seq+Adder* encoding. In those cases, the number of clauses generated is too large and exceeds the resource limits used in our evaluation.

The column #*Veri* under *Translation* shows results for VeriPB verification of the translation certificate from VeritasPBLib. Except for a few instances for *GTE* and *Seq+Adder* yielding large proofs, VeriPB is successful. If the translation check passes, then this guarantees that the CNF encoding does not remove any solutions of the pseudo-Boolean formula.

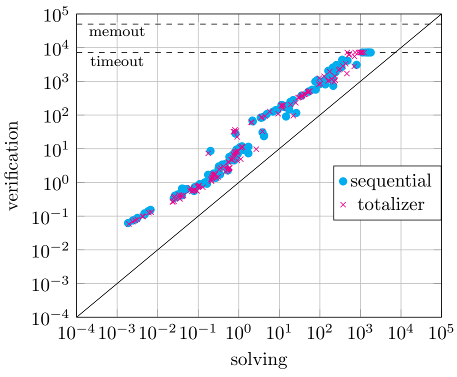
The columns #*Solved* and #*Verified* under *Solving* show how many instances can be solved by Kissat, and from those, how many can be verified by VeriPB. If a satisfiable formula is verified, then all clauses learned by Kissat are also valid for the original pseudo-Boolean formula, as is the satisfying assignment found. If an unsatisfiable formula is verified, then a correct proof of unsatisfiability for the PB formula has been produced.

We can verify 99% of the solved unsatisfiable instances, which shows that the current proof-of-concept approach is already practical in this setting. VeriPB proof logs can also be produced for satisfiable instances. These proof logs contain the derivations of all pseudo-Boolean constraints used in the solvers reasoning until a solution is found. Hence, it is possible to verify that the reasoning of the solver is sound, even if the instance is satisfiable. For satisfiable formulas we can verify that the reasoning was correct for 95% of the solved instances. However, even when VeriPB does not terminate within the time limit, we can still certify that the satisfying assignment found by the SAT solver is valid for the original PB formula. We note that there is still ample room for performance improvements in VeriPB—in particular, when it comes to verifying the *DRAT* proofs produced by the SAT solver, which do not even use pseudo-Boolean reasoning, but are simply clausal proofs syntactically rewritten in pseudo-Boolean format. Implementing backwards checking [GN03] and some minor engineering should get VeriPB close to the performance of DRAT-trim [WHH14] on *DRAT* proofs. Hence, there is no fundamental difficulty is improving the performance of VeriPB, but such work is mostly a matter of engineering and is fairly orthogonal to the contributions of this paper.

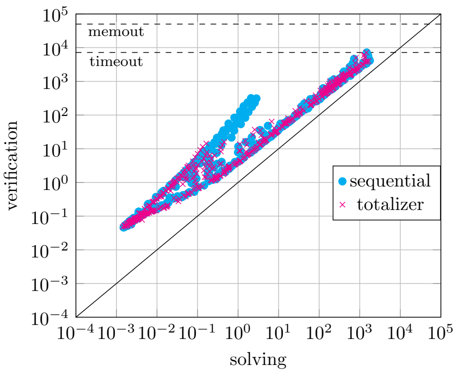
Figures 6 and 7 present the relationship between the end-to-end solving time (encoding the pseudo-Boolean formula to CNF and solving the resulting formula using Kissat) and the time to check the resulting proof log for the end-to-end solving using VeriPB. It can be seen that even though we can verify most instances, verification is often considerably slower than solving. The time to verify an instance in proportion to solving it varies significantly. This is due to the verification of the proof generated by the SAT solver, since VeriPB has not been optimized to check such proofs.

Figure 6 compares end-to-end solving and verification time for formulas with only cardinality constraints and Figure 7 does the same comparison for formulas with general pseudo-Boolean constraints. We split the benchmarks between satisfiable and unsatisfiable instances to analyze if the satisfiability of the formula affects the overhead of verification.

For the *sequential counter* encoding, verification of the proof for satisfiable instances takes on average 11.27 ± 6.98 times longer than solving and for unsatisfiable instances 18.30 ± 22.12 times longer. Even though verification times

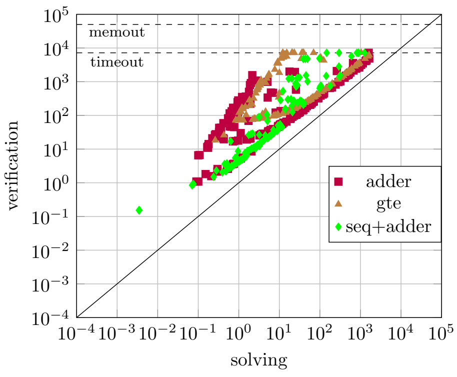


(a) *Satisfiable cardinality formulas.*

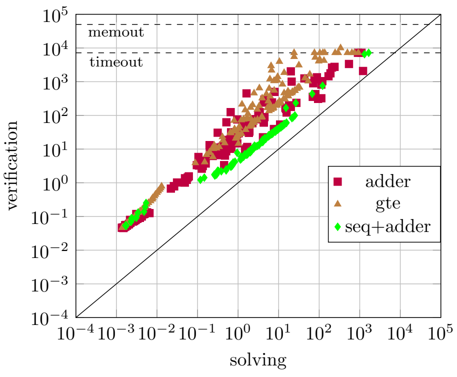


(b) *Unsatisfiable cardinality formulas.*

Figure 6: *Comparing end-to-end solving and verification time for cardinality formulas.*



(a) *Satisfiable general PB formulas.*

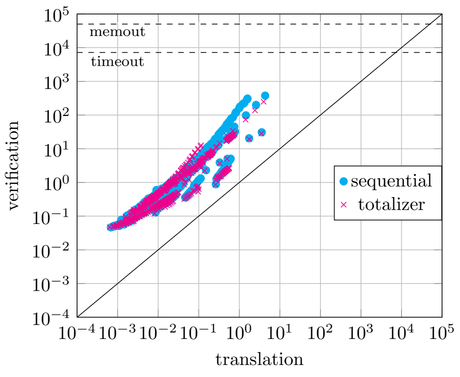


(b) *Unsatisfiable general PB formulas.*

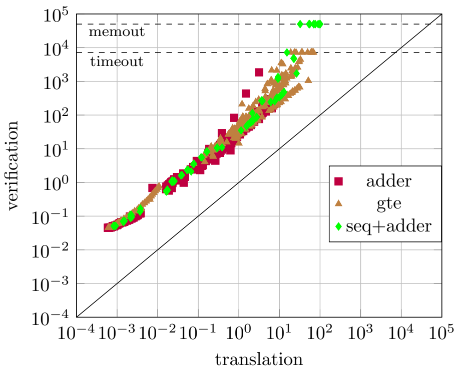
Figure 7: *Comparing end-to-end solving and verification time for pseudo-Boolean formulas.*

vary significantly for unsatisfiable instances, in the median satisfiable instances are checked within 8.45 times the solving time while unsatisfiable instances are checked within 5.16 times the solving time. The overhead for satisfiable instances may seem large, but note that VeriPB also checks if the derivations in the proof log of these instances are sound and not just the correctness of the result, which is the case in many occasions, e.g., the SAT competition.

We observe a similar behavior with the *totalizer* encoding, where verifying the proof takes on average 11.30 ± 8.38 times longer than solving for satisfiable instances and 14.83 ± 14.54 times longer than solving for unsatisfiable instances. Similarly, there are quite different verification times for unsatisfiable instances, in the median there is an 8.62 times overhead for satisfiable instances while only having an overhead of 5.17 times for unsatisfiable instances.



(a) *Cardinality formulas.*



(b) *General pseudo-Boolean formulas.*

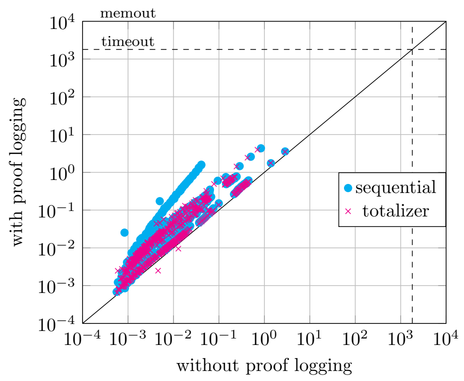
Figure 8: *Comparison between CNF translation and verification of proof logging.*

For the general pseudo-Boolean formulas, we observe a higher verification overhead with respect to solving time. In particular, *GTE* has an average overhead of 54.67 ± 61.85 times for unsatisfiable instances and 89.88 ± 134.77 times for satisfiable instances, with a median overhead of 36.71 times for unsatisfiable instances and 18.68 times for satisfiable instances. A similar scenario applies with the *Adder* encoding with an average 29.69 ± 28.41 times overhead for unsatisfiable instances and 54.00 ± 99.29 times for satisfiable instances, with a median overhead of 28.42 times for unsatisfiable instances and 5.44 times for satisfiable instances. Verifying the results reported by the SAT solver are harder for formulas containing PB constraints than for formulas containing cardinality constraints.

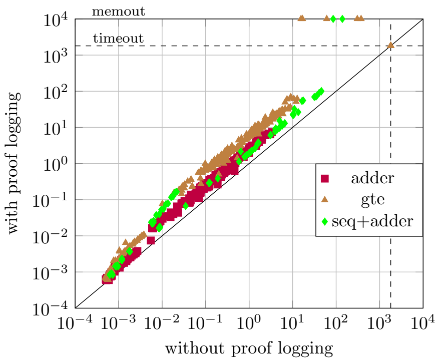
When considering formulas with both cardinality and pseudo-Boolean constraints, the observed overhead is smaller than for the other formulas with an average overhead of 7.89 ± 9.44 times for unsatisfiable instances and 13.11 ± 19.39 times for satisfiable instances, with a median overhead of 4.33 times for unsatisfiable instances and 5.21 times for satisfiable instances.

7.3 Translation and Verification

For a more detailed discussion of our results, let us first turn to the certified translation. Figure 8 compares the time for VeritasPBLib to generate the CNF translation and VeriPB to verify it. The verification overhead is far from negligible, but is not unreasonable. Over all encodings, for 75% of benchmarks verification takes at most 49 times longer than translation, and for 98% of benchmarks at most 100 times longer. While some overhead is natural, since the translation algorithm can just output a claimed proof while the verifier needs to perform the calculations to actually check it, our experiments do show that there is room for further improvements in efficiency both for the verifier and for the proof logging methods.



(a) *Cardinality formulas.*



(b) *General pseudo-Boolean formulas.*

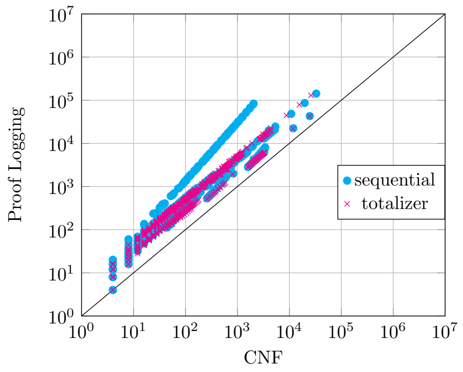
Figure 9: *Comparison of running times for CNF translation with and without proof logging.*

7.4 Overhead of Proof Logging

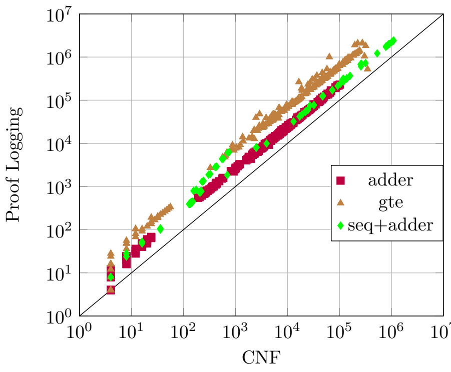
Figure 9 shows the overhead of proof logging when translating the pseudo-Boolean formulas to CNF. For the majority of the instances, the overhead is not too significant, and formulas with just cardinality constraints can still be translated under 10 seconds, while formulas with PB constraints can be translated under 100 seconds. The average overhead in running time for proof logging is a factor of 2–3 for all encodings except *GTE*, which incurs around a factor 5 in overhead. However, since translation is fast for the majority of instances, the additional overhead of proof logging is not an issue when translating the pseudo-Boolean formulas to CNF.

The proof logging overhead can be explained by the proofs being larger than the generated CNF formulas, as shown in Figure 10. For most instances the proof size seems to be within a constant factor of the CNF formula size, but for a collection of crafted vertex cover problems [EGNV18] the sequential counter encoding turns out to require proofs of super-linear size. These instances contain a cardinality constraint enforcing a constant fraction of the variables in the formula to be false, which is a worst-case scenario for the sequential counter encoding. While the number of clauses in the CNF translation and the number of proof steps are quadratic in the number of literals in the constraint, each reification step in the unary sum derivation in Proposition 2 introduces a constraint of linear size, making the total proof size cubic while the size of the CNF encoding remains quadratic. It would be desirable to find a more efficient derivation that only requires quadratic proof size in the number of literals in the constraint.

Additionally, there were 6 instances where VeritasPBLib had memory outs, as the whole proof for the translation is stored in memory. This could be improved in the future by only storing the proof for one constraint at a time in VeritasPBLib.



(a) *Cardinality formulas.*



(b) *General pseudo-Boolean formulas.*

Figure 10: *Comparison between CNF file size and proof logging file size in KiB.*

7.5 Comparison with PB Solvers

In Table 3, we report results on how VeritasPBLib together with Kissat used as a pseudo-Boolean solver using the sequential counter and adder encodings compares to state-of-the-art PB solvers, namely, MiniSat+ [ES06], Gurobi [Opt22] (version 9.5.1), NaPS [SN15] (version 1.02b), Open-WBO [MML14], RoundingSat [EN18] (commit b5de84d), and Sat4j [LP10] (version v20220212). All solvers were run with their respective default configurations.

The approach of VeritasPBLib+Kissat to transform a PB formula into CNF and using a CDCL SAT solver to solve the resulting formula is also made by other PB solvers such as MiniSat+, NaPS, and Open-WBO. However, the encodings used in these solvers differ, and VeritasPBLib uses a more recent SAT solver (Kissat). For this benchmark set, VeritasPBLib+Kissat outperforms other PB solvers in the *Card+PB* and *PB* categories while being third in the *Card* category. Note that for Sat4j, we ran the default version, which has native support for PB constraints and does not translate them to CNF but still uses a SAT solver to solve the resulting formula.[5] However, since Sat4j is written in Java and the underlying SAT solver is not as powerful as the other solvers, its performance is worse when compared to the other solvers.

Instead of using resolution like the SAT-based approaches, RoundingSat uses stronger pseudo-Boolean reasoning in the form of cutting planes. For this benchmark set, RoundingSat performed better than SAT-based solvers for the *Card* category but worse for the *Card+PB* and *PB* categories.

Figure 11 shows a cumulative plot with the runtime comparison of pseudo-Boolean solvers. We can observe that a majority of the instances are solved after

[5]Sat4j best-performing version for PB formulas is to run a cutting-planes-based solver with a resolution solver in parallel. We did not present results for this version since we only run single-threaded solvers, and this is not the default configuration of Sat4j. However, even with this version Sat4j consistently performs worse than other cutting-planes-based solvers like RoundingSat [EN18].

Table 3: *Number of solved instances by each PB solver*

Solver	Card	Card+PB	PB	Total
MiniSat+	490	269	323	1,082
Gurobi	610	256	230	1,096
NaPS	555	265	283	1,103
Open-WBO	600	275	316	1,191
RoundingSat	663	270	273	1,206
Sat4j	455	265	275	995
VeritasPBLib+Kissat	619	286	346	1,251

a few seconds. Overall, VeritasPBLib+Kissat not only provides certificates that can be checked by VeriPB, but is also one of the best approaches to solving pseudo-Boolean decision problems.

7.6 Certifying MaxSAT Optimal Values

Maximum Satisfiability (MaxSAT) [BJM21] is the optimization counterpart of SAT, where the goal is to maximize the number of satisfied clauses. The MaxSAT problem can be generalized to have *hard* and *soft* clauses, where hard clauses *must* be satisfied and soft clauses may or may not be satisfied. Each soft clause has a weight associated with it that corresponds to the cost of falsifying that soft clause. For the general MaxSAT problem, the optimization goal becomes to maximize the sum of the weights of the satisfied soft clauses. This optimization problem can also be viewed as minimizing the sum of the weights of falsified soft clauses. An optimal value of a MaxSAT formula corresponds to the minimal sum of the weights of the falsified soft clauses.

The annual MaxSAT Evaluation[6] focus on evaluating the current state-of-the-art in MaxSAT solvers. It has two main categories: (1) unweighted, where all soft clauses have a weight of 1, and (2) weighted, where soft clauses have a weight between 1 and 2^{63}. In contrast to the SAT competition, the results of MaxSAT solvers are not verified since there is no verification tool for MaxSAT. Instead, the optimal solution claimed by the solvers is checked to be a valid solution (i.e., satisfies all hard clauses, and the optimal value corresponds to the sum of the weights of the falsified soft clauses), and any of the competing solvers found no solution with a smaller value. However, this procedure does not give any correctness guarantees. It has occurred in previous years that a single solver found a (claimed) optimal solution for an instance, but this solution was later found not to be optimal.[7]

Even though VeritasPBLib+Kissat cannot be used to show the correctness of the solving procedure of a MaxSAT solver, it may be used to certify that the optimal value of a given instance is correct. Given a MaxSAT formula F and its respective

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

[7]`http://www.maxsat.udl.cat/15/results/index.html`

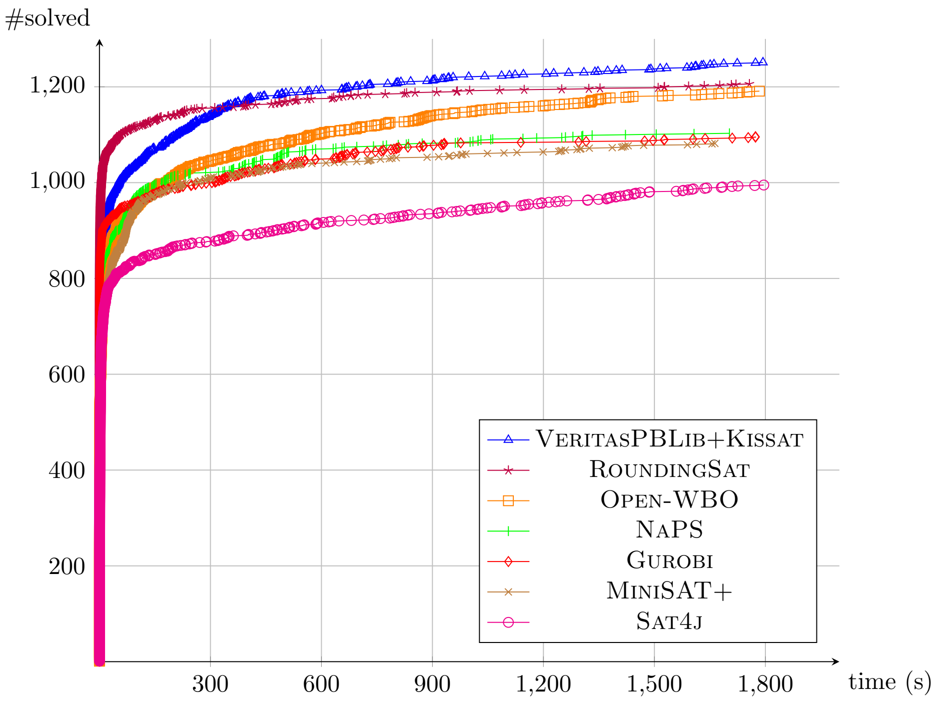


Figure 11: *Cumulative plot with runtime comparison of PB solvers.*

optimal value k, we turn the task of proving optimality of a MaxSAT instance F into solving a PB decision instance F_{PB} that encodes that no smaller optimal value exists for F. Let $F = F_h \cup F_s$ be a MaxSAT formula, where F_h represents h hard clauses and F_s represents s soft clauses. Let the weight associated with each soft clause $D_j \in F_s$ be a_j and k the optimal value of F. We construct F_{PB} as follows.

- Each clause $C = (\ell_1 \vee \ldots \vee \ell_n) \in F_h$ is added in pseudo-Boolean form to F_{PB}: $\ell_1 + \ldots + \ell_n \geq 1$;
- For each clause $D_j = (\ell_1 \vee \ldots \vee \ell_m) \in F_s$, we introduce a fresh variable b_j and add the clause in pseudo-Boolean form to F_{PB}: $\ell_1 + \ldots + \ell_m + b_j \geq 1$;
- Add a PB constraint to F_{PB} that restricts the sum of the weights of falsifying soft clauses to be at most $k-1$: $a_1 b_1 + \ldots + a_s b_s \leq k - 1$.

We can use VeritasPBLib to translate F_{PB} to CNF, use Kissat to solve the resulting formula, and then verify the results with VeriPB. If the formula is unsatisfiable, we can certify that there is no solution with an objective value smaller than k. The solution that results in the optimal value k has already been tested to be a valid solution. Therefore, if we prove that no better solution exists, we can show that the optimal value returned by the MaxSAT solver is correct.

Table 4: *Number of translated, solved and verified instances for each encoding when certifying the results from the MaxSAT Evaluation 2022.*

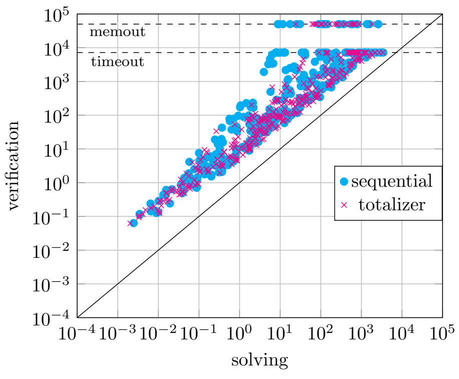
			Translation		Solving	
Category	#Inst	Encoding	#CNF	#Verified	#Solved	#Verified
Unweighted	468	Sequential	411	333	329	265
		Totalizer	448	408	358	307
Weighted	473	Adder	455	346	193	139
		GTE	262	186	221	169

To evaluate how effective VeritasPBLib+Kissat is to certify the results of the MaxSAT Evaluation 2022,[8] we used the instances for which at least one solver found an optimal solution, namely, 468 unweighted instances and 473 weighted instances. Similarly to the previous experiments, we used a memory limit of 15 GB, a time limit of 7,200 seconds for verifying the proof with VeriPB, and a time limit of 1,800 seconds for CNF translation with VeritasPBLib. We increase the SAT solving time for Kissat to 3,600 seconds to match the time limit used in the MaxSAT Evaluation.

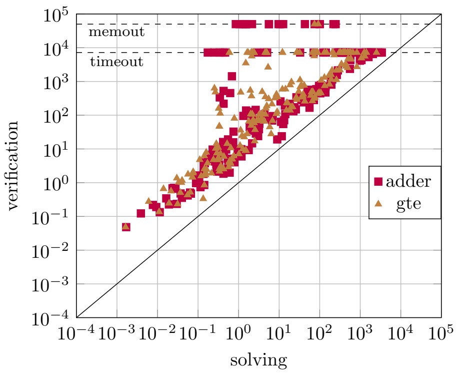
Table 4 shows the number of instances for which VeritasPBLib+Kissat could verify the optimal value. The information is split into *Translation* and *Solving*. The column #*CNF* presents the number of instances where VeritasPBLib successfully generated a CNF formula from the pseudo-Boolean formula F_{PB}. Note that all F_{PB} are unsatisfiable, and each of them only contains either a *single cardinality constraint* (in the case of unweighted) or a *single PB constraint* (in the case of weighted), with the remaining constraints being clauses. The column *Verified* under *Translation* shows how many instances were verified by VeriPB for the proof logging certificate generated by VeritasPBLib when translating each F_{PB} to CNF. The columns #*Solved* and #*Verified* under *Solving* present how many instances were solved by Kissat, and from those, how many were verified by VeriPB.

For the unweighted category, we can observe that 20 instances cannot be translated to CNF with the totalizer encoding, which shows the blowup in the CNF encoding when translating a single cardinality constraint to CNF. Overall, Kissat can solve 358 out of 468 instances (76%) using this approach. Even though this is a large percentage of instances, it does not match the performance of the best MaxSAT solvers, since they are able to prove optimality by solving a different CNF formula that is equivalent but simpler than the one we are solving with our approach. Nevertheless, by using VeritasPBLib+Kissat, we can certify 307 out of 358 instances (86%), which shows the positive result that if we can solve the formula with Kissat, we are likely able to certify the results. We can observe a similar behavior with the sequential counter encoding, albeit with worse performance, since this encoding is not as efficient for solving instances with a single large cardinality constraint.

[8]`https://maxsat-evaluations.github.io/2022/`



(a) *Unweighted MaxSAT.*



(b) *Weighted MaxSAT.*

Figure 12: *Comparison between end-to-end solving and verification time for the MaxSAT Evaluation 2022 instances that can be solved by* VeritasPBLib+Kissat*.*

For the weighted category, when using the GTE encoding, we can translate 262 out of 473 instances (55%) to CNF and when using the adder network encoding, we can translate 455 out of 473 instances (96%). This large difference is due to the exponential growth of the GTE encoding concerning the size of the weights, while the adder network encoding only grows linearly. However, despite this significant difference, Kissat can solve more instances with the GTE encoding than with the adder network encoding since the GTE encoding is arc consistent [Gen02] and the adder network encoding is not. When the formula can be translated to CNF using the GTE encoding, then Kissat can solve it in 221 out of 262 cases (84%). Overall, using VeritasPBLib+Kissat, we can certify 169 out of 221 instances (76%) when using the GTE encoding and 139 out of 193 instances (72%) when using the adder network encoding. If we consider instances that can be solved or certified by either using the adder network encoding or the GTE encoding, then Kissat can solve 247 instances, and VeriPB can certify 199 of those. This shows that the adder network encoding and GTE encoding are complementary and using both encoding can increase the number of certified instances.

Figure 12 compares the end-to-end time between translation plus solving with VeritasPBLib+Kissat and verifying the proof using VeriPB. As in Section 7.2, we can observe that although we can verify most instances that the SAT solver solves, verification is often considerably slower than solving the problem.

8 Concluding Remarks

In this work, we develop a general framework for certified translations of linear pseudo-Boolean constraints into CNF using cutting-planes-based proof logging. Since our method is a strict extension of the *DRAT* proof logging method used by conflict-driven clause learning (CDCL) SAT solvers, the proof for the PB-to-CNF translation can be combined with a SAT solver *DRAT* proof log to provide, for the first time, end-to-end verification for SAT-based pseudo-Boolean solvers. Our use of the cutting planes method is not only crucial to deal with the pseudo-Boolean format of the input, but the expressivity of the 0-1 linear constraints also allows us to certify the correctness of the translation to CNF in a concise and elegant way.

While there is still room for performance improvements in proof logging and verification, the experimental evaluation shows that our approach is feasible in practice. We believe that the generality of our method, which expresses the proof logging steps in terms of simple operations on a graph representation of the PB-to-CNF translation, is an important aspect of our work. To demonstrate this generality of our framework we show how to do proof logging for the sequential counter, binary adder and (generalized) totalizer encodings. We are optimistic that our framework can also be used for the watchdog encoding [BBR09], which builds on top of the totalizer encoding. It is less clear whether the graph representation can also be used in an elegant way to capture some of the *sorting networks* encodings found to be particularly efficient in [ES06], such as the *odd-even merge sorters* [Bat68] used in MiniSat+, or BDD-based encodings [Bry86, ES06], or whether more ad-hoc pseudo-Boolean proof logging methods would be needed for such encodings.

As discussed already in the introduction, our paper does not quite reach the goal of certifying *equivalence* of the original pseudo-Boolean formula F and the CNF translation F'. In one direction, it is clear that as long as F' is derived from F using cutting planes with reification, any satisfying assignment α to F yields a unique extended assignment $\alpha' \supseteq \alpha$ satisfying F' by giving all newly introduced variables the values determined by the reification rules (5a)–(5b). In the other direction, however, we do not formally establish that the CNF translation F' is as strong as the original pseudo-Boolean formula F in the sense that any satisfying assignment α' for F' is guaranteed to also satisfy F. As a quick technical detour, one way of achieving such guarantees would be, after having derived all clauses in F', to erase all constraints in F using the "checked deletion" rule in [BGMN22]. This is certainly doable in principle, but we currently know of no clean and simple way to formalize this in our graph-based translation framework. This is therefore another problem that we have to leave as future research.

Our work on proof logging for PB-to-CNF translations has also uncovered some technical questions that, to the best of our knowledge, have not been studied in the literature before, but would seem to merit further investigations. A common theme is that these questions revolve around possible trade-offs between encoding strength and encoding size, as explained below.

In our proofs of correctness for the order encoding, the derivation of binary clauses enforcing $z_i \geq z_{i+1}$ play a key role in the derivations, but are not included in the final CNF translation. This is a little bit surprising, since it would seem that such clauses would improve propagation and hence potentially help the SAT solver discover more facts. On the other hand, it is not clear how the presence of such clauses in the solver trail would affect the conflict analysis. Thus it would be interesting to study whether including such clauses in the PB-to-CNF translation would affect the SAT solving process in any systematic way.

Another question concerns the translation of cardinality constraints. When given a constraint $\sum_{i=1}^{n} a_i \ell_i \geq k$ such that $\sum_{i=1}^{n} a_i - k < k$, the PB-to-CNF translation instead uses the equivalent constraint $\sum_{i=1}^{n} a_i \overline{\ell_i} \leq \sum_{i=1}^{n} a_i - k$ because it introduces fewer auxiliary variables. On the other hand, it seems that the presence of auxiliary variables encoding partial information about constraints is precisely what allows SAT-based pseudo-Boolean solvers to compete with, and not seldom outperform, cutting-planes-based solvers [EGNV18]. And perhaps there could be a reason why the problem at hand was encoded with a greater-than-or-equal constraint rather than less-than-or-equal. It would seem relevant to investigate if there is a trade-off here between propagation strength (potentially leading to more efficient SAT solver search) and encoding size (potentially slowing down the solver due to the increased number of auxiliary variables).

A final question regarding CNF translation of circuits is whether it is better to encode propagations in both directions or only in one direction. If a circuit encodes the evaluation of a PB constraint $\sum_i a_i \ell_i \geq A$, then one direction of propagation is that any assignment to the literals ℓ_i making the constraint true should make the output gate of the circuit evaluate to true. The other direction of propagation is that any literal assignment violating the constraint should make the output gate of the circuit evaluate to false. In our proofs of correctness we generate constraints encoding both types of propagation, but it seems that in the final CNF translation it is most common to include only clauses enforcing one of the directions. This cuts the encoding size in half, but at the price of losing propagation power of the encoding. It would be quite interesting to investigate how enforcing two-way propagation or only one-way propagation affects the efficiency of the SAT solver search.

Concluding this section, we wish to emphasize that we view certified translations to CNF of pseudo-Boolean decision problems as only a first step. In the conference version of this paper, we expressed optimism that the techniques developed in this work should also be possible to extend to *core-guided MaxSAT solving* [FM06, MHL$^+$13], including proof logging support for derivation of clauses added during core extraction and objective function reformulation, and such results have very recently been announced in [BBN$^+$23]. While designing efficient proof logging for other MaxSAT approaches such as *implicit hitting sets (IHS)* [DB11] and *abstract cores* [BBP20] seems more challenging, we are hopeful that our work could lead to a unified proof logging method for all modern MaxSAT solving techniques, and also for pseudo-Boolean optimization using cutting-planes-based reasoning as in [DGN21, DGD$^+$21, EN18, LP10, SBJ21, SBJ22].

Acknowledgements

The authors wish to acknowledge helpful and stimulating discussions with Bart Bogaerts and Ciaran McCreesh. We are particularly grateful to Bart for sharing the manuscript [VWB22] using a very elegant reification technique that we wish we would have thought of, and we believe it would be worth exploring whether similar ideas could be used in our framework to improve the efficiency of verification. We are also thankful for discussions and feedback at the workshops *Pragmatics of SAT* and *Proof eXchange for Theorem Proving* in 2021 and the conference *SAT* in 2022, where a preliminary version of this work was presented. We would like to extend a special thanks to the *SAT '22* and *JAIR* anonymous reviewers for a wealth of comments and questions that helped to improve this manuscript considerably.

This work has been financially supported by the Swedish Research Council grant 2016-00782, the Independent Research Fund Denmark grant 9040-00389B, the National Science Foundation award CCF-1762363, the Amazon Research Award, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[AGJ+18] 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.

[Bar95] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the Spring Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS '68)*, volume 32, pages 307–314, April 1968.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BB21] Lee A. Barnett and Armin Biere. Non-clausal redundancy properties. In *Proceedings of the 28th International Conference on Automated Deduction (CADE-28)*, volume 12699 of *Lecture Notes in Computer Science*, pages 252–272. Springer, July 2021.

[BBH22] Randal E. Bryant, Armin Biere, and Marijn J. H. Heule. Clausal proofs for pseudo-Boolean reasoning. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction* *and Analysis of Systems (TACAS '22)*, volume 13243 of *Lecture Notes in Computer Science*, pages 443–461. Springer, April 2022.

[BBHJ13] Adrian Balint, Anton Belov, Marijn JH Heule, and Matti Järvisalo. Proceedings of sat competition 2013: Solver and benchmark descriptions, 2013.

[BBN$^+$23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided MaxSAT solving. Submitted manuscript, March 2023.

[BBP20] Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, July 2020.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BFFH20] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFHH$^+$17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, 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 *LNCS*, pages 220–236. Springer, 2017.

[CFMSSK17] Luís Cruz-Filipe, Joao 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 *LNCS*, pages 118–135. Springer, 2017.

[CGS17] 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.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*

(CP '11), volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, September 2011.

[DGD$^+$21] 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.

[DGN21] 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*.

[EG21] Leon Eifler and Ambros Gleixner. 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)*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, May 2021.

[EGMN20] 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.

[EGNV18] Jan Elffers, Jesús Giráldez-Cru, Jakob Nordström, and Marc Vinyals. Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers. 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 75–93. Springer, July 2018.

[EN18] 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.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. 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 252–265. Springer, August 2006.

[Gen02] Ian P. Gent. Arc consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[GMM⁺20] 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.

[GMN20] 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.

[GMN22] 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.

[GMNO22] Stephan Gocht, Ruben Martins, Jakob Nordström, and Andy Oertel. Experimental repository for "Certified CNF translations for pseudo-Boolean solving". Available at `https://doi.org/10.5281/zenodo.6610581`, June 2022.

[GN03] 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.

[GN21] 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.

[GSD19] 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.

[HHW13a] 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.

[HHW13b] 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.

[HMS12] Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-Boolean constraints into SAT. In *Proceedings of KI 2012: Advances in Artificial Intelligence, the 35th Annual German Conference on AI*, volume 7526 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KB21] Daniela Kaufmann and Armin Biere. AMulet 2.0 for verifying multiplier circuits. 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 357–364. Springer, March-April 2021.

[KBBN22] Daniela Kaufmann, Paul Beame, Armin Biere, and Jakob Nordström. Adding dual variables to algebraic reasoning for circuit verification. In *Proceedings of the 25th Design, Automation and Test in Europe Conference (DATE '22)*, pages 1435–1440, March 2022.

[KFB20] Daniela Kaufmann, Mathias Fleury, and Armin Biere. The proof checkers Pacheck and Pastèque for the practical algebraic calculus. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 264–269, September 2020.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[MHL$^+$13] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João P. Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, October 2013.

[MML14] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. Open-WBO: A modular MaxSAT solver. 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 438–445. Springer, July 2014.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[Opt22] Gurobi Optimization. Gurobi Optimization. Available at `https://www.gurobi.com/`, 2022.

[PS15] Tobias Philipp and Peter Steinke. PBLib – a library for encoding pseudo-Boolean constraints into CNF. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT '15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer, September 2015.

[RBK$^+$18] Daniela Ritirc, Armin Biere, Manuel Kauers, A Bigatti, and M Brain. A practical polynomial calculus for arithmetic circuit verification. In *3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2 '18)*, pages 61–76, 2018.

[RM16] Olivier Roussel and Vasco M. Manquinho. Input/output format and solver requirements for the competitions of pseudo-Boolean solvers. Revision 2324. Available at `http://www.cril.univ-artois.fr/PB16/format.pdf`, January 2016.

[SBJ21] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-Boolean optimization by implicit hitting sets. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:20, October 2021.

[SBJ22] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. 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)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, August 2022.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume

3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[SN15] Masahiko Sakai and Hidetomo Nabeshima. Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE Transactions on Information and Systems*, 98-D(6):1121–1127, June 2015.

[Van08] 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`.

[VWB22] Dieter Vandesande, Wolf De Wulf, and Bart Bogaerts. Qmaxsatpb: A certified maxsat solver. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, 2022.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.

[WHH14] 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.

Certified Core-Guided MaxSAT Solving

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.

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 [IP01, CIP09]. 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 [BHvMW21], constraint programming (CP) [RvBW06], and mixed integer programming (MIP) [AW13, BR07]. 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

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.

results [CKSW13, AGJ+18, GSD19, BMN22]. 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* [ABM+11, MMNS11], 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* [GN03], *TraceCheck* [Bie06], *DRAT* [HHW13a, HHW13b], *GRIT* [CMS17], and *LRAT* [CHH+17] 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 [BJM21, LM21], 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 [BS97, MS99, MMZ+01], for MaxSAT there is a rich variety of approaches including *linear SAT-UNSAT* (or *model-improving search*) [ES06, LP10, PRB18], *core-guided search* [FM06, NB14, ADR15, AG17], *implicit hitting set (IHS)* search [DB13a, DB13b], and some recent work on branch-and-bound methods [LXC+22] (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 [BLM07, LNOR11, MM11, MIB$^+$19, FMSV20, PCH20, PCH21, PCH22, IBJ22], none of them comes close to providing general-purpose proof logging, because they apply only for very specific algorithm implementations and/or fail to capture the full range of reasoning used in an algorithmic approach. A recent work [VDB22] by two co-authors on the current paper instead leverages the pseudo-Boolean proof logging system VeriPB [Ver] to certify correctness of the unweighted linear SAT-UNSAT solver QMaxSAT. VeriPB is similar in spirit to *DRAT*, but operates with more general 0–1 linear inequalities rather than just clauses. This simplifies reasoning about optimization problems, and also makes it possible to capture the powerful MaxSAT solver inferences in a more concise way. VeriPB has previously been used for proof logging of enhanced SAT solving techniques [GN21, BGMN22] and pseudo-Boolean solving [GMNO22], as well as for providing proof-of-concept tools for a nontrivial range of techniques in constraint programming [EGMN20, GMN22] and subgraph solving [GMN20, GMM$^+$20].

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 [IBJ21], based on RC2 [IMM19], and show how to enhance CGSS to output proofs of correctness of its reasoning, including sophisticated techniques such as stratification [ABGL12, MAGL11], intrinsic-at-most-one constraints [IMM19], hardening [ABGL12], weight-aware core-extraction [BJ17], and structure sharing [IBJ21]. 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 [EG21, GMM$^+$20, KM21] 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 [BN21, GN21, GMNO22]. 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* [Bar95]; 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 [CCT87], 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 $C\upharpoonright_\rho$ 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 [GN21, BGMN22]. 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*

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

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

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 $C^{\Rightarrow}_{\mathrm{reif}}(z)$ for (1a) and $C^{\Leftarrow}_{\mathrm{reif}}(z)$ 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, [LBJ20, BJM21]. 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 [AKMS12, MDM14], 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 $lits(O_{orig})$ 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., $lits(K) \subseteq lits(O_{orig})$, and (ii) F_{orig} implies K, which means that any solution to F_{orig} has to set at least one literal in $lits(K)$ to true. The *cost* $w(K, O) = \min\{coeff(O, \ell) :$

$\ell \in lits(K)\}$ 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

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

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 $y_{K,k} \Leftarrow \sum_{b \in lits(K)} b \geq k$ 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 [MJML14]. 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 [ABGL12, MAGL11], hardening [ABGL12], the intrinsic-at-most-ones technique [IMM19], weight-aware core extraction [BJ17], and structure sharing [IBJ21].

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 $\sum_{b \in \mathcal{S}} \bar{b} \leq 1$ 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 $\ell_{\mathcal{S}} + \sum_{b \in \mathcal{S}} \bar{b} \geq 1$ 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 $b \in lits(K)$ is lowered and the lower bound LB is increased by $w(K, O_{ref})$. 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 $O_{orig} \geq O_{ref}$ 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 $O_{orig}(\alpha) < UB$ is obtained, it is logged in the proof, which adds the *objective-improving constraint*

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

(which is

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

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 [BGMN22] 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 [KP18, KP19, BB03]. The two solvers we consider use totalizers [BB03], 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 $lits(\eta)$ 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 $S_\eta = \{y_{\eta,1}, \ldots, y_{\eta,|lits(\eta)|}\}$, 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 $0 \leq \alpha \leq |lits(\eta_1)|$, $0 \leq \beta \leq |lits(\eta_2)|$, and $0 \leq \sigma \leq |lits(\eta)|$. We use the notational conventions in (4a)–(4b) that $y_{\ell,1} = \ell$ for all leaves ℓ, and that $y_{\eta,0} = 1$ and $y_{\eta,|lits(\eta)|+1} = 0$ for all nodes η (so that clauses containing $y_{\eta,0}$ or $y_{\eta,|lits(\eta)|+1}$ 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 $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta,j})$ and $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta,j})$ 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 $C_\eta^{\Leftarrow}(\alpha,\beta,\sigma)$ and $C_\eta^{\Rightarrow}(\alpha,\beta,\sigma)$ in (4a)–(4b) can be derived from the constraints $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta,\sigma})$, $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta,\sigma})$, $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta_1,\alpha})$, $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta_1,\alpha})$, $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta_2,\beta})$, and $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta_2,\beta})$ by standard cutting planes derivations as in [VDB22]. 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 $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta^*,\sigma})$ and $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta^*,\sigma})$ 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 $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta,j})$ and $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta,j})$ 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 $C_{\mathrm{reif}}^{\Leftarrow}(y_{\eta,j})$ and $C_{\mathrm{reif}}^{\Rightarrow}(y_{\eta,j})$ 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 [GMNO22] 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

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

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., [CCT87]).

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 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 $b \in lits(O_{ref})$ for which $UB < LB + coeff(O_{ref}, b)$ 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 UB_{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., $UB_{est} \geq UB$), hardening steps based on UB_{est} can be justified by first deriving $O_{orig} \leq UB_{est} - 1$, 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.

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), $\perp$

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 [CGSa, IBJ21], which uses the OLL algorithm and structure-sharing totalizers. We employed VeriPB [Ver], 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 [Max22], 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 [BBN+23].

Overhead of Proof Logging. To evaluate the overhead in solver running time, we compared the standard CGSS solver [CGSb] 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,

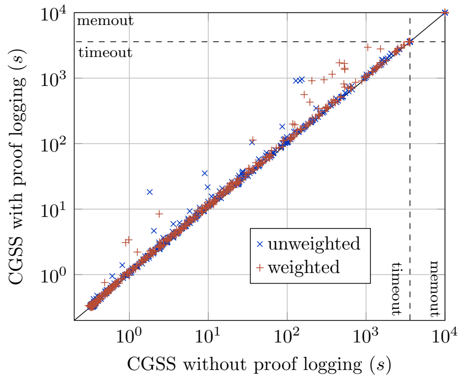


Figure 1: *Running time of CGSS with and without proof logging.*

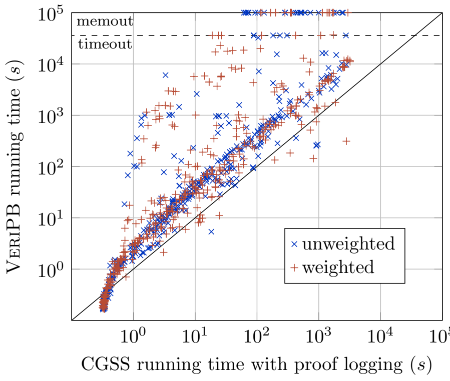


Figure 2: *CGSS running time compared to time required for proof checking.*

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 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.

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	$9e_3 + \mathbf{8y_{1,3}} + 2y_{1,5} + y_{2,2} + 2y_{2,3} + 2y_{3,2} + 7y_{4,2} + 2y_{5,2} + o_1 + o_2 + 24$

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 $w(K_{new}, O_{new})$ rather than $w(K_{old}, O_{old})$. 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 $F = \left\{\left(\bigvee_{i=1}^{5} b_i\right), (o_1 \vee o_2)\right\} \cup \{b_i \vee e_i \mid i = 1, \dots, 5\}$ 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 [IBJ21] 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 [GN21, BGMN22], 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 [HHW13a]. 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 [PRB18]) and implicit hitting set (IHS) search (such as MaxHS [DB13a, DB13b]). 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 [DGD$^+$21] or IHS [SBJ21, SBJ22], and perhaps even to similar techniques in constraint programming [GBDS20] and answer set programming [AKMS12].

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

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP ’12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[ADR15] Mario Alviano, Carmine Dodaro, and Francesco Ricca. A MaxSAT algorithm using cardinality constraints of bounded size. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI '15)*, pages 2677–2683. AAAI Press, 2015.

[AG17] Carlos Ansótegui and Joel Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, 2017.

[AGJ+18] 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.

[AKMS12] Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP '12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 211–221, September 2012.

[AW13] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. In Michael Jünger and Gerhard Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer, 2013.

[Bar95] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, January 1995.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BBN+23] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Experimental repository for "Certified core-guided MaxSAT solving". `https://doi.org/10.5281/zenodo.7709687`, May 2023.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJ17] Jeremias Berg and Matti Järvisalo. 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)*, volume 10416 of *Lecture Notes in Computer Science*, pages 652–670. Springer, August 2017.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. 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*, pages 929–991. IOS Press, 2nd edition, February 2021.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.

[BMN22] Bart Bogaerts, Ciaran McCreesh, and Jakob Nordström. Solving with provably correct results: Beyond satisfiability, and towards constraint programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`, August 2022.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BR07] Robert Bixby and Edward Rothberg. 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, February 2007.

[BS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CGSa] Certifying version of the CGSS core-guided MaxSAT solver with structure sharing. `https://gitlab.com/MIAOresearch/software/certified-cgss`.

[CGSb] 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/`.

[CHH⁺17] 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.

[CIP09] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Revised Selected Papers from the 4th International Workshop on Parameterized and Exact Computation (IWPEC '09)*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, September 2009.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CMS17] 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.

[DB13a] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DB13b] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP '13)*, volume 8124 of *Lecture Notes in Computer Science*, pages 247–262. Springer, 2013.

[DGD⁺21] 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.

[EG21] Leon Eifler and Ambros Gleixner. 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), volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, May 2021.

[EGMN20] 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.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. 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 252–265. Springer, August 2006.

[FMSV20] Yuval Filmus, Meena Mahajan, Gaurav Sood, and Marc Vinyals. MaxSAT resolution and subcube sums. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 295–311. Springer, July 2020.

[GBDS20] Graeme Gange, Jeremias Berg, Emir Demirović, and Peter J. Stuckey. 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)*, volume 12296 of *Lecture Notes in Computer Science*, pages 205–221. Springer, September 2020.

[GMM$^+$20] 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.

[GMN20] 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.

[GMN22] 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.

[GMNO22] 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.

[GN03] 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.

[GN21] 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.

[GSD19] 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.

[HHW13a] 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.

[HHW13b] 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.

[IBJ21] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Refined core relaxation for core-guided MaxSAT solving. In *27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João P. Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, September 2019.

[IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, March 2001. Preliminary version in *CCC '99*.

[KM21] Sonja Kraiczy and Ciaran McCreesh. 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)*, pages 1396–1402, August 2021.

[KP18] Michal Karpinski and Marek Piotrów. Competitive sorter-based encoding of PB-constraints into SAT. In *Proceedings of Pragmatics of SAT*, volume 59 of *EPiC Series in Computing*, pages 65–78. EasyChair, 2018.

[KP19] Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3-4):234–251, 2019.

[LBJ20] Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete maxsat solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 347–354. IOS Press, 2020.

[LM21] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2021.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, 2011.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[LXC$^+$22] Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Boosting branch-and-bound MaxSAT solvers with clause learning. *AI Communications*, 35(2):131–151, 2022.

[MAGL11] João Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4):317–343, 2011.

[Max22] MaxSAT evaluation 2022. `https://maxsat-evaluations.github.io/2022`, August 2022.

[MDM14] António Morgado, Carmine Dodaro, and João P. Marques-Silva. Core-guided MaxSAT with soft cardinality constraints. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 564–573. Springer, September 2014.

[MIB$^+$19] António Morgado, Alexey Ignatiev, María Luisa Bonet, João P. Marques-Silva, and Samuel R. Buss. DRMaxSAT with MaxHS: First contact. 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 239–249. Springer, July 2019.

[MJML14] Ruben Martins, Saurabh Joshi, Vasco M. Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Proceedings of the 20th International Conference on Principles and Practice of Constraint Programming (CP '14)*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer, September 2014.

[MM11] António Morgado and João Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI '11)*, pages 924–926, 2011.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[NB14] Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI '14)*, pages 2717–2723. AAAI Press, 2014.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. 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)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference* *on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] 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.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[SBJ21] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-Boolean optimization by implicit hitting sets. In *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP '21)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 51:1–51:20, October 2021.

[SBJ22] Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. 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)*, volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, August 2022.

[VDB22] 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.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability

Abstract

Proof logging has long been the established method to certify correctness of Boolean satisfiability (SAT) solvers, but has only recently been introduced for SAT-based optimization (MaxSAT). The focus of this paper is solution-improving search (SIS), in which a SAT solver is iteratively queried for increasingly better solutions until an optimal one is found. A challenging aspect of modern SIS solvers is that they make use of complex "without loss of generality" arguments that are quite involved to understand even at a human meta-level, let alone to express in a simple, machine-verifiable proof.

In this work, we develop pseudo-Boolean proof logging methods for solution-improving MaxSAT solving, and use them to produce a certifying version of the state-of-the-art solver Pacose with VeriPB proofs. Our experimental evaluation demonstrates that this approach works in practice. We hope that this is yet another step towards general adoption of proof logging in MaxSAT solving.

1 Introduction

Thanks to tremendous progress over the last decades on algorithms for combinatorial search and optimization, today *NP*-hard problems are routinely solved in many practical applications. Unfortunately, as these algorithms get more and more sophisticated, it also gets more and more challenging to avoid errors sneaking in during

Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesand. "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.

algorithm design and implementation. It is well-known that modern combinatorial solving algorithms in different paradigms can sometimes produce “solutions” that violate hard constraints, claim that suboptimal solutions are optimal, or declare that feasible problems lack solutions [BBN+23, BB09, BLB10, CKSW13, GSD19, JHB12].

Although there are many ways to address this problem, including software testing techniques such as fuzzing [BB09, PB23], and design of formally verified software [Fle20], the most promising approach appears to be the use of *certifying algorithms* [ABM+11, MMNS11] with so-called *proof logging*. What this means is the algorithm should not only produce an answer, but also a *proof* that this answer is correct. Such proofs should follow simple rules, as specified by a formal *proof system*, so that they can easily be verified by an independent *proof checker*. In addition to guaranteeing correctness, proof logging brings many other advantages: it enables advanced *testing* (since one can detect correct answers found for invalid reasons, and also test instances for which the answer is not known), detailed *debugging* (since invalid proof steps pinpoint where errors happened), *auditability* (since proofs can be stored and verified independently of which algorithm was used), and *performance analysis* (since proofs can be mined for insights on which reasoning steps were crucial for reaching the final conclusion).

Proof logging has been particularly successful in the domain of Boolean satisfiability (SAT) solving [BHvMW21], where a large variety of proof systems has seen the light of day [BCH21, Bie06, GN03, WHH14]. Using proof logging has long been mandatory in the main track of the SAT competitions, and it is hard to overestimate the impact this has had on improving overall correctness and reliability of SAT solvers. This has stimulated the spread of proof logging into other combinatorial solving paradigms, including SAT modulo theories (SMT) [SFBF21, BRK+22], automated planning [EH20, ERH17, ERH18, Rög17], and mixed integer linear programming [EG23, DEGH23].

Proof Logging for MaxSAT Solving In view of the above discussion, it is interesting to compare the developments in other combinatorial optimization paradigms to the state of affairs in maximum satisfiability (MaxSAT), the optimization version of the SAT problem. Without loss of generality, MaxSAT can be described as the problem of maximizing a linear objective O subject to satisfying a Boolean formula F in conjunctive normal form (CNF). Although MaxSAT is arguably the one optimization paradigm closest to SAT, and although several proof systems for formalizing MaxSAT reasoning have been proposed [BLM07, LNOR11, MM11, PCH20, PCH21, PCH22], for a long time there has been no practically feasible proof logging method for state-of-the-art MaxSAT solvers. This changed only recently when pseudo-Boolean proof logging using VeriPB [GN21, BGMN23] was proposed for MaxSAT [Van23, VDB22], a proposal that was followed by the successful design and implementation of VeriPB proof logging for modern core-guided MaxSAT solvers [BBN+23].

In this paper, we revisit proof logging work for *solution-improving search (SIS)* [Van23, VDB22], also referred to as *model-improving search* or *linear SAT-UNSAT*

(LSU) search, and consider state-of-the-art solving techniques. In the SIS approach—which is much simpler to explain than, e.g., core-guided [FM06] or implicit hitting set [DB13] search—a SAT solver is repeatedly called on the formula F, each time with an added *solution-improving constraint* asking for increasingly better solutions with respect to the objective O, and the problem turns infeasible when the last solution found was optimal. In the work by Vandesande et al. [Van23, VDB22], the main technical challenge was to certify correctness of the CNF encodings of these solution-improving constraints, which could then essentially be concatenated with the proof logging generated by the SAT solver (modulo some non-trivial engineering).

At first sight, it seems that implementing pseudo-Boolean proof logging in a state-of-the-art MaxSAT solver using solution-improving search would mostly be a matter of carefully transferring already developed techniques [Van23, VDB22], perhaps combining them with proof logging ideas developed for other CNF encodings [GMNO22]. After all, the distinguishing feature of a top-of-the-line SIS solver is the choice of CNF translation for reasoning about the objective function, such as, in the case of Pacose, the *polynomial watchdog (DPW)* encoding [BBR09]. Once proof logging for such a CNF encoding is in place, it seems reasonable to expect that the rest should be plain sailing.

It is all the more surprising, then, that it turns out nothing could be further from the truth. To minimize the time the MaxSAT solver spends on generating PW encodings, an essential step is to introduce completely unconstrained variables that can be used to perform different comparisons with a single CNF encoding; this is referred to as the *dynamic polynomial watchdog encoding (DPW)* [PRB18]. Loosely speaking, if we know that the best possible objective value lies in the range $[lo, hi]$, then instead of generating repeated encodings $O \geq V$ to probe different possible objective values V in this range, one can introduce free variables t_i encoding a tare sum T taking values between 0 and $hi - lo$ and try to maximize the value $T = T^*$ for which one single DPW-encoded constraint $O - T \geq lo$ holds. Once the maximum T^* has been found, it is clear that $O = lo + T^*$ is the best possible objective value, since without loss of generality T could be set to any value. But how can such a meta-argument be expressed in simple propositional logic reasoning?

In what follows, we provide a brief, if still high-level, discussion of some of the challenges that arise when trying to design simple proofs to certify such fairly complex "without loss of generality" arguments, and then outline how such challenges can be overcome.

Solution-Improving "Without Loss of Generality" Reasoning As already discussed above, the key aspect in which different solution-improving MaxSAT solvers differ is how they encode the solution-improving constraints. In order to compute the value of a linear expression L over 0–1 variables of interest, Pacose uses the polynomial watchdog encoding to describe a Boolean circuit BC with output variables z_k such that $z_k = 0$ implies $L \geq 1 + k \cdot 2^P$ (for some fixed integer P). If we chose L to be the objective function O that we are maximizing, this would allow to find the interval $\left[1 + k^* \cdot 2^P, (k^* + 1) \cdot 2^P\right]$ in which the optimal value lies by calling the SAT solver with the prechosen partial assignment $z_k = 0$ (referred to as an *assumption*) for increasing values of k until the solver returns that there is no satisfying assignment. To determine the exact location of the optimum in this interval, additional, completely unconstrained, variables t_i, called *tare variables*, are used to encode an integer $T = \sum_{i=0}^{P-1} 2^i t_i$ in the range $\left[0, 2^P - 1\right]$. The actual circuit in the encoding uses the linear form $L = O - T$, so that $z_k = 0$ means $O - T \geq 1 + k \cdot 2^P$. By making SAT solver calls with suitable assumptions on the unconstrained t_i-variables, the optimal value of the objective function can be computed.

Given the CNF encoding of a circuit $\mathsf{BC}(O - T \geq 1 + k \cdot 2^P)$ evaluating the inequality $O - T \geq 1 + k \cdot 2^P$ as outlined above, the solution-improving search proceeds in two phases:

1. The *coarse convergence phase* identifies the largest k for which $z_k = 0$ is possible.

2. The *fine convergence phase* then maximizes the tare variable sum T.

Let us discuss this process in slightly more detail, and explain why it presents challenges from a proof logging point of view.

If during the coarse convergence phase a SAT solver call with assumption $z_k = 0$ returns a satisfying assignment α achieving objective value at least $1 + k \cdot 2^P$, the solver stores the information $z_k = 0$ (in the form of a unit clause $\overline{z}_k$), which enforces that any future solutions found have to be at least this good. The SAT solver is then called again with $z_{k'} = 0$ for some $k' > k$ to probe whether a solution exists with value at least $1 + k' \cdot 2^P$. Here it is relevant to note that fixing $z_k = 0$ could remove assignments corresponding to optimal solutions. For instance, if the optimal value is $V = V^* + 1 + k \cdot 2^P$, this value could be achieved by an assignment α' setting $T = T^* > V^* + 1$. For such an α' we would have $O - T = -T^* + V^* + 1 + k \cdot 2^P \leq k \cdot 2^P$, which would violate $z_k = 0$. However, since the tare variables are unconstrained, in this case there would also exist another assignment α'' achieving objective value $V^* + k \cdot 2^P$ for which $T = 0$, and so it is safe to require that solutions improving on α should satisfy $z_k = 0$.

In the fine convergence phase the z_k-variables are all fixed, and assumptions on the tare variables are made in the SAT solver calls to determine the exact value of the optimal solution. This again relies on reasoning without loss of generality, claiming that one can always choose $T \geq s$ for any value $0 \leq s < 2^P$. But now we are treading on dangerous ground: clearly, we cannot assume both $T = 0$ and $T \geq s > 0$ simultaneously! How can we convince ourselves, and more importantly, how can we convince a proof checker, that our derivations are consistent? At a meta-level, we can argue that since the tare variables are completely unconstrained in the original encoding, we should be able to fix them to any value we like at any given point in time. But how do we produce a simple, machine-verifiable proof that this is sound? And are we even sure this is sound?

Discussion of Our Contribution In this work, we show how pseudo-Boolean proof logging with VeriPB [GN21, BGMN23] can certify correctness of the complex CNF encodings used in state-of-the-art solution-improving MaxSAT solvers, as well as of the subtle without loss of generality reasoning applied on these encodings. To give a sense of how this can be done, we need to give a high-level description how VeriPB proofs work (referring the reader to later sections for the missing technical details).

A VeriPB proof maintains a set of *core constraints* $\mathcal{C}$, initialized to the formula F, together with a set of *derived constraints* $\mathcal{D}$ inferred by the solver. The proof semantics ensures that $\mathcal{C}$ and F have the same optimal value for O and that any solution to $\mathcal{C}$ can be extended to $\mathcal{D}$. A new constraint C can be derived "without loss of generality" by the *redundance-based strengthening rule*, which requires the explicit specification of a substitution ω (mapping variables to truth values or literals) together with explicit proofs

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{1}$$

that all consequences on the right (with the substitution ω applied to the constraints) follow from previously derived constraints $\mathcal{C} \cup \mathcal{D}$ together with the negation $\neg C$ of the constraint to be inferred. This guarantees that if some assignment α satisfies everything so far but violates C, the "patched" assignment $\alpha \circ \omega$ satisfies also C and does not worsen the objective.

To make our informal discussion simple and concrete, suppose that we have a CNF encoding of a circuit $\mathsf{BC}(O - T \geq lo)$ evaluating $O - T \geq lo$, and that the solver has derived no constraints but only has the input formula F. If we want to fix $T = T^*$ using the redundance rule (1), we would have to find a substitution ω such that $F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{T \neq T^*\}$ implies $\left(F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{T = T^*\}\right)\restriction_\omega$. But it seems like this would force us to prove that if we take any assignment satisfying the Boolean circuit and modify the value of some of its inputs (the tares), the circuit would remain satisfied, and this is just not true. So although the redundance-based strengthening rule is very strong, it is not clear how it can be used to argue that the tare variables are unconstrained.

We get around this problem by first deriving a copy *shadow circuit* BC' of the original circuit, but substituting fixed values t_i^* for the tare variables, so that $\mathsf{BC}'(O - T^* \geq lo)$ evaluates $O - T^* \geq lo$. We then let ω be the substitution setting $t_i = t_i^*$ for all i and mapping all other variables x in BC to the corresponding shadow variables x' in BC', so that, effectively, the shadow circuit computes the substitution needed. This turns our application of the redundance rule (1) into

$$F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T \neq T^*\} \tag{2a}$$

$$\vdash \left(F \cup \{\mathsf{BC}(O - T \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T = T^*\}\right)\restriction_\omega \cup \{O\restriction_\omega \geq O\} \tag{2b}$$

$$= F \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{\mathsf{BC}'(O - T^* \geq lo)\} \cup \{T^* = T^*\} \cup \{O \geq O\} \tag{2c}$$

(where the final line (2c) is simply the result of applying the substitution ω to (2b)). If we study (2c) carefully, we see that all we need to prove about the circuit now is that the two copies of the shadow circuit in the consequences are implied by the same shadow circuit in the premises, and so (2c) follows trivially from the premises (2a).

This idea of using shadow circuits is crucial for certifying the correctness of assigning tare variables without loss of generality. However, we need to get rid of the completely unrealistic assumption that the solver would not have learned any constraints in $\mathcal{D}$. This is a problem in that the above argument fails when such learned constraints $D \in \mathcal{D}$ contain variables in the BC-circuit, since then there is no way to prove $D\restriction_\omega$ as required in (1).

Here a second idea discovered in recent VeriPB development turns out to be very helpful. Very briefly, it can be shown that if in the proof we enforce the requirement that all new constraints D derived by strengthening are immediately moved to the core set $\mathcal{C}$, referred to as *strengthening-to-core*, then the redundance rule (1) can be simplified to

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\}, \tag{3}$$

omitting the proof obligations for the derived set $\mathcal{D}$. This means that we can ignore the problems arising from derived constraints when using shadow circuit reasoning.

We stress that this is only a brief and informal discussion that sweeps many technical challenges under the rug. Perhaps one of the most annoying such challenges is that the tare variables are sometimes fixed one at a time, and then a new shadow circuit is required for every new fixing. It would be desirable to find better ways of dealing with this problem.

We have implemented our methods in the state-of-the-art solution-improving MaxSAT solver Pacose [PRB18] to make it output VeriPB proofs, and have performed an extensive evaluation of how such proof logging works in practice. While there is certainly room for performance improvements in both proof generation and proof checking, the significance of our contribution is that we present practical methods to certify correctness for a solving paradigm that has previously been beyond the reach of proof logging. We hope that our work can serve as an impetus towards general adoption of proof logging for MaxSAT, and can stimulate further research on how to make these proof logging techniques more efficient.

As a final remark, we note that an interesting aspect of recent progress in proof logging is that it brings together all three software quality assurance methods discussed in the opening paragraphs above. While proof logging does seem like the most viable approach to certify correctness in combinatorial solving, extensive use of fuzzing techniques has been instrumental in our work to debug both proof logging routines and the VeriPB proof checker. This fuzzing, in turn, relies on the use of proof logging and on feedback from the proof checker. Finally, although we do not address this aspect in the current paper, formally verified proof checking backends as in [GMM+24, IOT+24] are crucially needed to ensure that the verdict of proof checkers for increasingly powerful proof logging systems can be trusted.

Outline of This Paper After reviewing some preliminaries in Section 2, we discuss the dynamic polynomial watchdog (DPW) encoding in Section 3. In Section 4 we describe how to design proof logging for solution-improving solvers using the DPW encoding, including a discussion of possible variations of our method (and of why simply using SAT proof logging for the final unsatisfiability call does not work). We report results from an empirical evaluation in Section 5 and end with some conclusions and a discussion of future research directions in Section 6.

2 Preliminaries

In this section, we review some pseudo-Boolean basics and then discuss MaxSAT in general and solution-improving search in particular, referring the reader to [BN21, LM21, BJM21] for more details.

Pseudo-Boolean Constraints and Proofs We write x to denote a $\{0,1\}$-valued Boolean variable, and write $\overline{x}$ as a shorthand for $1-x$, using ℓ to denote such *positive* and *negative literals*, respectively. A (linear) *pseudo-Boolean (PB) constraint* C is a 0–1 integer linear inequality $\sum_i w_i\ell_i \geq A$. Without loss of generality, we will often assume our constraints to be *normalized*, meaning that all literal are over distinct variables and the coefficients w_i and the *degree* A are non-negative. A *PB formula* is a conjunction of PB constraints.

A (disjunctive) *clause* is a PB constraint $\sum_i \ell_i \geq 1$ with all coefficients and degree equal to 1. We sometimes refer to constraints $\ell \geq 1$ with a single literal as *unit clauses* ℓ. We say that a formula is in *conjunctive normal form (CNF)* if it is a conjunction of clauses. A *(linear) pseudo-Boolean term* is a weighted sum $\sum_i w_i\ell_i$ of literals with integer coefficients. A *(partial) assignment* α is a (partial) function from variables to $\{0,1\}$; it is extended to literals by respecting the meaning of negation. We write $C\!\upharpoonright_\alpha$ for the constraint obtained from C by substituting all assigned variables x by $\alpha(x)$ (and simplifying). A constraint C is *satisfied* under α if $\sum_{\alpha(\ell_i)=1} w_i \geq A$, and a formula F is satisfied if all its constraints are. We say that F *implies* C, denoted $F \models C$, if all assignments that satisfy F also satisfy C.

A *pseudo-Boolean optimization* (PBO) instance consists of a formula F and a linear term $O = \sum_i w_i\ell_i$ (called the *objective*). An assignment α to the variables in F and O that satisfies F is a *solution* to the instance, which is optimal if it *maximizes* the value $O\!\upharpoonright_\alpha = \sum_i w_i\alpha(\ell_i)$.[1] For a PBO instance (F, O) the VeriPB proof system maintains a *proof configuration* of *core* and *derived constraints* $(\mathcal{C}, \mathcal{D})$, initialized to F and $\emptyset$, respectively. The VeriPB proofs we consider are in the so-called *strengthening-to-core* mode, which maintains the invariant that all constraints in the

[1]Note that most of the PBO literature is formulated in terms of *minimization*, and this is also the perspective of VeriPB, but reasoning in terms of maximization is in line with the papers on solution-improving MaxSAT relevant for this work. We therefore adopt this perspective here, although the actual VeriPB proofs will argue in terms of minimizing the negation of the objective as described here.

derived set $\mathcal{D}$ are implied by the core set $\mathcal{C}$. Constraints can be moved from $\mathcal{D}$ to $\mathcal{C}$ but not vice versa. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ and added to $\mathcal{D}$ using the *cutting planes* proof system [CCT87] as follows:

Literal Axioms. For any literal ℓ_i, $\ell_i \geq 0$ is an axiom.

Linear Combination. Given two previously derived PB constraints C_1 and C_2, any positive integer linear combination of these constraints can be inferred.

Division. Given the normalized PB constraint $\sum_i w_i \ell_i \geq A$ and a positive integer c, the constraint $\sum_i \lceil w_i / c \rceil \ell_i \geq \lceil A/c \rceil$ can be inferred.

Some additional VeriPB proof rules extending cutting planes are as listed below—we refer to [BGMN23, GN21, HOGN24] for more details. For optimization problems we have rules for improvements of or rewriting of the objective function:

Objective Improvement. Given a total assignment α that satisfies $\mathcal{C} \cup \mathcal{D}$, one can add the constraint $O \geq 1 + O\restriction_\alpha$ to $\mathcal{C}$, which forces the search for strictly better solutions.

Objective Reformulation. The current objective O can be replaced by a new objective O_{new} given explicit proofs from the core set $\mathcal{C}$ (using the VeriPB proof rules above) of the constraints $O - O_{\text{new}} \geq 0$ and $O_{\text{new}} - O \geq 0$ (i.e., a proof that $O = O_{\text{new}}$ holds).

Importantly, there are also rules for deriving non-implied constraints as long as the optimal value of the objective is preserved. VeriPB has a generalization of the RAT rule [JHB12] that makes use of *substitutions* ω, mapping variables to truth values or literals (where we extend the meaning of $C\restriction_\omega$ to denote C with each x replaced by $\omega(x)$):

Redundance-Based Strengthening. The constraint C can be inferred and added to $\mathcal{C}$ by explicitly specifying a substitution ω and proofs $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \{C\})\restriction_\omega \cup \{O\restriction_\omega \geq O\}$. This assumes strengthening-to-core mode—otherwise derivations for all constraints in $\mathcal{D}\restriction_\omega$ are also needed (but then C can be placed in $\mathcal{D}$ instead of $\mathcal{C}$).

Intuitively, this rule shows that ω remaps any solution of $\mathcal{C}$ that does not satisfy C to a solution of $\mathcal{C}$ that satisfies also C without worsening the objective value. A typical use case of redundance-based strengthening is *reification*, which is the derivation of two pseudo-Boolean constraints that encode $\ell \Leftrightarrow D$ for some PB constraint D and for some fresh literal ℓ.

Finally, VeriPB has rules for deleting constraint in a way that guarantees that no spurious better-than-optimal solutions are introduced:

Deletion A constraint $D \in \mathcal{D}$ in the derived set can be deleted at any time. If strengthening-to-core mode is used, then deleting a constraint $C \in \mathcal{C}$ in the core set requires an explicit proof that C is implied by $\mathcal{C} \setminus \{C\}$. Otherwise, it is sufficient to show the weaker property that C can be derived from $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening.

MaxSAT, Incremental SAT Solving, and Solution-Improving Search An instance of (weighted partial) Maximum Satisfiability (MaxSAT) consists of a CNF formula F and a pseudo-Boolean objective $O = \sum_i w_i \ell_i$ to be maximized under satisfying assignments to F, where we can assume without loss of generality that all literals in O are over distinct variables and that the constants are positive. Viewing MaxSAT in terms of an objective function and a CNF formula is equivalent to the more classical definition in terms of hard and soft clauses, in the sense that maximizing the objective corresponds to maximizing the total weight of satisfied soft clauses (see, e.g., [LBJ20] for more details).

The solution-improving search (SIS) algorithm we focus on in this work makes extensive use of incremental SAT solving with assumptions [ES03]. Invoking a SAT solver on a CNF formula F with a set of *assumptions* $\mathcal{A}$, i.e., a partial assignment, returns either 1. SAT and an extension of $\mathcal{A}$ that satisfies F or 2. UNSAT if no such assignment exists.

Given a MaxSAT instance (F, O), solution-improving search (SIS) computes an optimal solution by issuing a sequence of queries to a SAT solver asking for solutions of improving quality until an optimal one is found. More precisely, during search SIS maintains the best known solution α^*. In each iteration, the algorithm queries a SAT solver on the working formula $F \wedge \mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$, where $\mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$ is a CNF formula that is satisfied by an assignment α if and only if it is a better solution than α^*, i.e., if $O{\restriction_{\alpha}} > O{\restriction_{\alpha^*}}$. If the SAT solver returns SAT, a better solution has been obtained and the working formula updated accordingly. Otherwise, if the SAT solver reports UNSAT, the best known solution α^* is determined to be optimal and the search is terminated.

The existing practical instantiations of SIS differ mainly in how the encoding of the formula $\mathsf{AsCNF}(O > O{\restriction_{\alpha^*}})$ is realized. Numerous CNF encodings of pseudo-Boolean constraints have been proposed for this task [ES06, JMM15, KP19, MPS14, Sin05]. For many instantiations of SIS the main challenge for proof logging is to certify the clauses added when encoding the objective constraint [VDB22, Van23], but as we will explain in the rest of this paper the so-called Dynamic Polynomial Watchdog encoding requires much more subtle arguments.

3 The Dynamic Polynomial Watchdog Encoding for SIS

The polynomial watchdog (PW) encoding [BBR09] is currently one of the best approaches for encoding pseudo-Boolean constraints in CNF, in terms of being compact while still propagating well. Using it for solution-improving search requires some non-trivial alternations, however, such as the addition of a dynamic constant. In this section we review this dynamic polynomial watchdog (DPW) encoding to the extent required for MaxSAT solution-improving search (SIS), referring the reader to [PRB18] for more details.

3.1 Initialization

Given a linear pseudo-Boolean term $L = \sum_i w_i \ell_i$, we define $w_{\max}$ to be the largest constant appearing in L. Additionally, we let $P := \lfloor \log_2(w_{\max}) \rfloor$ be one smaller than the number of bits in the binary representation of $w_{\max}$ and $W := \sum_i w_i$ be the maximum value for L. The polynomial watchdog encoding for L is a CNF formula $\mathsf{PW}(L)$ with $c := \lceil \frac{W}{2^P} \rceil$ output variables z_k for $k \in [0, c-1]$ enforcing the implications $\overline{z}_k \Rightarrow L \geq 1 + k \cdot 2^P$. In words, a satisfying assignment α of $\mathsf{PW}(L)$ that sets $\alpha(z_k) = 0$ will also satisfy $\sum_i w_i \alpha(\ell_i) \geq 1 + k \cdot 2^P$. We describe the formula $\mathsf{PW}(L)$ in more detail in Section 4.1.

Example 1. Consider a MaxSAT instance (F, O) and a working formula $F^w = F \wedge \mathsf{PW}(O)$. Assume we first invoke a SAT solver on F^w under the assumption $z_{k-1} = 0$ and then a second time under the assumption $z_k = 0$, and that the solver reports SAT for the first call and UNSAT for the second. At this point, we know that an optimal solution α^{opt} has value $O{\restriction}_{\alpha^{\text{opt}}}$ in the range $\left[1 + (k-1) \cdot 2^P, k \cdot 2^P\right]$.

The PW encoding was proposed as a way of enforcing a fixed bound B on the term L by considering a (static) constant $T = B - (1 + k \cdot 2^P)$, where k is the largest integer for which $B \geq 1 + k \cdot 2^P$, and encoding $\mathsf{PW}(L - T)$ [BBR09]. Then a solution that sets the kth output z_k of $\mathsf{PW}(L-T)$ to 0 will also satisfy $\sum_i w_i \alpha(\ell_i) - T \geq 1 + k \cdot 2^P$, which is equivalent to $\sum_i w_i \alpha(\ell_i) \geq B$. The dynamic polynomial watchdog (DPW) encoding [PRB18] is an extension of the PW encoding that allows dynamically changing the value of T, and therefore also of B, so that the optimal value can be determined precisely with a single CNF encoding.

Consider a MaxSAT instance (F, O) and let $P = \lfloor \log_2(w_{\max}) \rfloor$ as described above. Instantiations of SIS with DPW introduce a “dynamic constant” in the form of a *tare* term $T := \sum_{i=0}^{P-1} 2^i \cdot t_i$, for fresh variables t_i not appearing anywhere else in the instance. The SAT solver is instantiated with the working formula $F \wedge \mathsf{PW}(O - T)$. Now we can use the output variables z_k to determine the optimal value within an additive constant 2^P, and then assign the tare T to values in $\left[0, 2^P - 1\right]$ to determine the precise value in that range. These are the *coarse convergence* and *fine convergence* phases mentioned in Section 1, which we describe in more detail next.

3.2 Coarse Convergence Phase

During the initial coarse convergence phase, only assumptions over the output variables z_k are made. Whenever a solution α is found, a call to the SAT solver is made with the assumption $z_k = 0$ where k is the largest natural number such that $O{\restriction}_\alpha \geq 1 + (k-1) \cdot 2^P$. The coarse convergence phase ends when the solver reports UNSAT. The following observation summarizes the relevant conclusions of coarse convergence.

Observation 1. *Assume F is satisfiable and the SAT solver returns UNSAT under an assumption $z_{k^*} = 0$ in the coarse convergence phase. Then 1. there is a solution α^* to*

$F \wedge \mathsf{PW}(O - T)$ that assigns the tare variables so that $(O - T)\restriction_{\alpha^*} \geq 1 + (k^* - 1) \cdot 2^P$ holds, and 2. no solution β to F assigning also the tare variables can satisfy $(O - T)\restriction_{\beta} \geq 1 + k^* \cdot 2^P$.

In words, coarse convergence provides bounds on the maximum value of $O - T$ obtainable by any solution of F. Importantly, as the tare term T is unconstrained by the formula F, its value can without loss of generality be assumed to be 0 at this stage, resulting in bounds on the objective value of optimal solutions as well. From now on, the algorithm commits to only searching for solutions that have $O - T$ in the specified interval, adding the unit clauses $\overline{z}_{k^*-1}$ and z_{k^*} to the working formula before proceeding to the fine convergence phase. In practice, whenever the SAT solver returns SAT after being called with assumption $\overline{z}_k$, the unit clause $\overline{z}_k$ is added immediately, allowing the SAT solver to simplify its clause database.

3.3 Fine Convergence Phase

During the fine convergence phase, assumptions for the tare variables are used to pinpoint the precise optimal value. Let k^* be the value for which the assumption $z_{k^*} = 0$ returned UNSAT in coarse convergence, and $o^* = O\restriction_{\alpha^*}$ the objective value of the currently best known solution α^*. Then we define $s := o^* - (k^* - 1) \cdot 2^P$ to be the smallest value of the tare that would force an improved solution. The next call to the SAT solver assumes $t_i = 1$ for all tare variables for which the i^{th} bit in the binary representation of s is 1. These assumptions enforce $T \geq s$, so any solution α to the working formula (which now includes the unit clause $\overline{z}_{k^*-1} \geq 1$) that extends the assumptions will satisfy $O\restriction_{\alpha} \geq o^* + 1$.

The fine convergence phase continues in this manner until the SAT solver reports UNSAT, at which point an optimal solution has been found. As the value of s is monotonically increasing, we add unit clauses t_i to the working formula whenever we have deduced that the i^{th} bit t_i in the tare T can safely be set to 1 in any solution (and hence in any future SAT call), which is the case when $s - 1 \geq 1 + \sum_{j=i}^{P-1} 2^i \cdot t_j$ holds. The fact that we have $s - 1$ rather than s in this last inequality is related to *stratification*, which we discuss next.

3.4 Stratification

Stratification is a technique for partitioning the indices of an objective $O = \sum_{i=1}^{m} w_i \ell_i$ into two sets $\{H, L\}$ in a way that allows computing the maximum values first of $O_H = \sum_{i \in H} w_i \ell_i$ and then of $O_L = \sum_{i \in L} w_i \ell_i$, and finally combining them to get the maximum value of O.

Specifically, stratification is applied when $\gcd\{w_i \mid i \in H\} \geq \sum_{i \in L} w_i$, i.e., when the greatest common divisor of the coefficients in O_H is at least the sum of all coefficients in O_L. SIS with the DPW encoding and stratification will first run coarse and fine convergence only on O_H as described above. At the end of the fine convergence, the SAT solver returns UNSAT after being invoked with assumptions that enforce $T_H \geq s$ for the tare term T_H added to the DPW encoding of O_H and some constant s. At this stage, the value of T_H will be fixed to $s - 1$ with unit clauses, effectively fixing O_H to its maximum value. This fixing of O_H is consistent with the unit clauses learned in the previous section. After this O_L is optimized via coarse and fine convergence under the fixed value of O_H. The solution obtained at the end of the final fine convergence phase will be optimal with respect to the original instance. For more details on stratification, we refer the reader to [ALM09, PRB21].

Example 2. Consider the objective $O = 10x_1 + 5x_2 + 5x_3 + 3x_4 + 2x_5$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5\}$. Since $\gcd\{10, 5, 5\} = 5 \geq 3 + 2$, changes of the objective restricted to $\{x_1, x_2, x_3\}$ will dominate any contributions from $3x_4 + 2x_5$. If a solution α with $O_H{\upharpoonright_\alpha} = 15$ is found, we can without loss of generality assume $O_H \geq 15$, since for any solution β with $O_H{\upharpoonright_\beta} < 15$ we have $O{\upharpoonright_\beta} \leq O{\upharpoonright_\alpha}$. Notice that maximizing first O_H and then O_L can remove some optimal solutions from the search space, but never all of them.

4 Certifying Solution-Improving MaxSAT with the DPW Encoding

We are now ready to describe how to do proof logging for solution-improving MaxSAT with the dynamic polynomial watchdog encoding. In addition to certifying the correctness of CNF encodings, as done in previous work on proof logging SIS for MaxSAT [VDB22, Van23], we need to certify the without loss of generality reasoning discussed in Section 3. This turns out to require quite intricate proof logging methods.

We start with a brief discussion how to certify the DPW encoding. We then turn to proof logging for the without loss of generality reasoning during the coarse and fine convergence phases. Afterwards, we deal with proof logging for stratification. We defer a discussion of minor additional heuristics used in state-of-the-art solvers to Appendix B. We note that for all clauses learned by the SAT solver we can use standard VeriPB proof logging, and since all such learned clauses are logically implied by the working formula it is safe to add them to the derived set $\mathcal{D}$. This means that we can ignore all constraints added to the database by the SAT solver when we perform redundance-based strengthening steps.

4.1 Proof Logging for Clauses of the DPW Encoding

Figure 1 depicts the structure of the DPW encoding of the term $2x_1 + 3x_2 + 5x_3 + 7x_4$. For a term L in which the largest coefficient has P bits, the encoding introduces P totalizers [BB03] (which are circuits that sort their inputs), and $P - 1$ mergers. The i^{th} totalizer takes as input all variables in L for which the corresponding coefficient has its i^{th} bit equal to 1.

Proof logging for the DPW encoding boils down to taking care of the totalizer encodings as described in [VDB22]. At a high level, the proof for $\mathsf{PW}(O - T)$ derives a number of constraints encoding implications $y \Rightarrow C_y$ and $y \Leftarrow C_y$, where y are variables in the auxiliary variable set Y and C_y are suitably chosen PB constraints

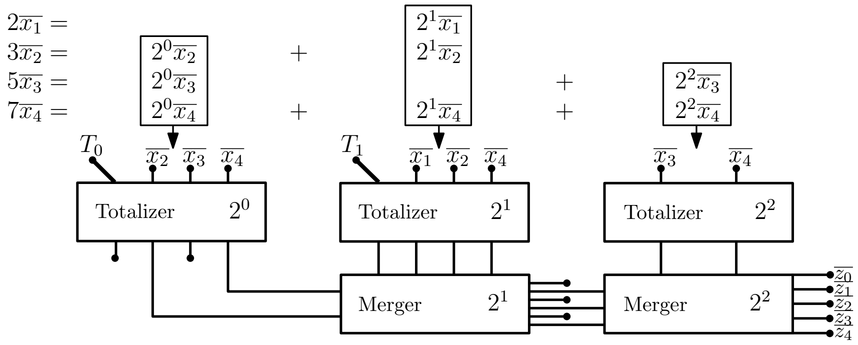

$2\overline{x_1} =$
$3\overline{x_2} =$
$5\overline{x_3} =$
$7\overline{x_4} =$
$2^0\overline{x_2}$ $2^0\overline{x_3}$ $2^0\overline{x_4}$
+
+
$2^1\overline{x_1}$ $2^1\overline{x_2}$ $2^1\overline{x_4}$
+
+
$2^2\overline{x_3}$ $2^2\overline{x_4}$
T_0
T_1
Totalizer 2^0
Totalizer 2^1
Totalizer 2^2
Merger 2^1
Merger 2^2
$\overline{z_0}$ $\overline{z_1}$ $\overline{z_2}$ $\overline{z_3}$ $\overline{z_4}$

Figure 1: *Illustration of the polynomial watchdog encoding.*

over the variables in $O - T$. A concrete example is the output variable z_k for which the constraint C_{z_k} is chosen as $O - T \leq k \cdot 2^P$. From these pseudo-Boolean definitions all clauses in the CNF encoding added to the solver database can be derived with explicit VeriPB derivations. A technical point that is crucial for the proof logging is that in this way we only need to add the PB definitions of new variables to the core set C. The clauses actually used for the SAT solver calls are implied from these definitions, and can therefore be placed in the derived set $\mathcal{D}$.

4.2 Proofs Without Loss of Generality Using Shadow Circuits

The MaxSAT solving algorithm uses without loss of generality (wlog) reasoning when 1. introducing fresh variables for encoding $\mathsf{PW}(O - T)$; 2. adding unit clauses $\overline{z}_k$ during coarse convergence; 3. learning unit clause over the tare variables t_i during fine convergence; and 4. concluding that the optimal value has been found.

To see why unit clauses $\overline{z}_k \geq 1$ require wlog reasoning, suppose in the coarse convergence phase that the SAT solver returns a solution α when invoked with the assumption $z_k = 0$, indicating that $(O - T)\restriction_\alpha \geq 1 + k \cdot 2^P$. The constraint $\overline{z}_k \geq 1$ is *not* entailed by the solution-improving constraint $O \geq O\restriction_\alpha$, since some other (possibly optimal) solution β might have $O\restriction_\beta \geq O\restriction_\alpha$ but assign the tare variables so that $(O - T)\restriction_\beta < 1 + k \cdot 2^P \leq (O - T)\restriction_\alpha$ holds. However, since the tare variables are not constrained by the original formula F, any solution to F could be extended to any fixed value for the tare T. Hence, in particular, we can assume without loss of generality that $T = 0$, which in turn implies that $\overline{z}_k \geq 1$.

The fine convergence phase makes use of the fact that the DPW encoding does not constrain T, which takes values in the range $\left[0, 2^P - 1\right]$. The unit clauses $t_i \geq 1$ learned are not entailed, but can be deduced since the tare variables are unconstrained in the DPW encoding. This requires a VeriPB proof that wlog $T \geq s - 1$. When the SAT solver reports UNSAT during fine convergence, it does so under the assumption that a specific set of tare variables take value 1. If this yields UNSAT, then we can conclude that the current solution is optimal (since we can wlog assume T to be equal to the value that led to UNSAT).

It is worth noticing that the without loss of generality arguments above are quite intricate even at a human meta-level. The coarse convergence phase repeatedly claims to be able to assume $T = 0$, after which the fine convergence phase picks an increasing sequence $0 < s_1 < s_2 < \ldots$ and assumes $T \geq s_i - 1$ wlog. Finally, a specific value $T = s_{i^*}$ is used to argue about optimality. The meta-level argument for why this works is that no conclusions are drawn from the assumptions made during coarse and fine convergence that invalidate subsequent assumptions. The challenge is how to convince a mechanical proof checker of this.

Consider first proof logging for the coarse convergence phase, and suppose the solver returns SAT when invoked with assumption $\overline{z}_k$. The only rule that would allow us to derive $\overline{z}_k \geq 1$ without loss of generality (from the argument that we can set $T = 0$ wlog) is *redundance-based strengthening*, which requires specification of a witness substitution ω that can be used to "patch" any assignment α in which $\overline{z}_k \geq 1$ is violated. More formally, our witness should guarantee that $\mathcal{C} \cup \mathcal{D} \cup \{\neg(\overline{z}_k \geq 1)\} \models \big(\mathcal{C} \cup \{\overline{z}_k \geq 1\}\big)\!\restriction_\omega \cup \{O \leq O\!\restriction_\omega\}$. A natural approach would be to choose a witness ω that maps 1. z_k to 0, 2. all original variables to themselves, and 3. T to 0. Such a witness would make $(\overline{z}_k \geq 1)\!\restriction_\omega$ trivially true and would incur no proof obligations for the formula F or the objective O. However, setting $T = 0$ will not work for the constraints $C \in \mathcal{C}$ defining variables in the DPW encoding. If we fix $T = 0$, then we also need to update all auxiliary variables Y in the circuit evaluating $\mathsf{PW}(O - T)$. But how this should be done depends on which assignment α we need to patch, and the redundance rule has no mechanism for defining "conditional witnesses" $\omega = \omega(\alpha)$.

To determine how the witness should assign the auxiliary variables in $\mathsf{PW}(O-T)$, we devise a new proof logging technique that we call *shadow circuits*. Corresponding to each auxiliary variable y defined as the reification of a PB constraint C_y in the original circuit, a *shadow circuit for a fixed value v* has a fresh variable $y^{T=v}$ defined by $y^{T=v} \Leftrightarrow C_y\!\restriction_{T \mapsto v}$. In words, the defining constraints of $y^{T=v}$ and y are the same except that we fix the tare variables t_i so that $T = v$. The definitions of such shadow circuits are stored in the core set $\mathcal{C}$ since they are derived using the redundance rule. Note that the shadow circuit only "copies" the pseudo-Boolean definitions of the variables and not their clausal encodings.

Shadow circuits provide us with a mechanism to compute witnesses for the redundance rule that allow us to assume the value of T and certify the without loss of generality reasoning. During coarse convergence, each addition of a constraint $\overline{z}_i \geq 1$ is logged with a witness that maps all tare variables t_i to 0 and other auxiliary variables y in $\mathsf{PW}(O - T)$ to their counterparts $y^{T=0}$ in the shadow circuit for $T = 0$. During fine convergence, the constraints $T \geq s - 1$ are derived using shadow circuits for $s - 1$, which allows adding unit constraints over individual tare variables to the proof. Finally, for proving optimality a shadow circuit for the final value s^* for which the SAT solver returned UNSAT will be used to derive contradiction.

The next proposition gives a more formal summary of the wlog proof logging performed during the coarse convergence phase. The proof for this proposition, together with precise descriptions of the other wlog proof logging steps, are given in Appendix A.

Proposition 2. *Suppose the VeriPB proof log contains derivations of reification constraints $\overline{z}_k \Leftrightarrow O - T \geq 1 + k \cdot 2^P$ and a shadow circuit for $T = 0$ as well as the constraint $O \geq 1 + k \cdot 2^P$. Then the constraint $\overline{z}_k \geq 1$ can be derived using redundance-based strengthening with witness $\omega = \{t_i \mapsto 0 \mid 0 \leq i \leq P - 1\} \cup \{y \mapsto y^{T=0} \mid y \in Y\}$.*

The constraint $O \geq 1 + k \cdot 2^P$ in Proposition 2 can be obtained by weakening the solution-improving constraint $O \geq O\restriction_\alpha + 1$ for the previously found solution α. If stratification is used, deriving $O_H \geq 1 + k \cdot 2^P$ requires more work (see Section 3.4 for details).

Our technique with shadow circuits and repeated without loss of generality arguments selecting (different) values for the same variables in T heavily relies on that VeriPB proofs in the *strengthening-to-core* mode maintain the guarantee that all constraints in the derived set $\mathcal{D}$ are entailed by the core set $\mathcal{C}$. In particular, what this means is that whenever we want to apply redundance-based strengthening, fixing tare variables and using the corresponding shadow circuit, we do not need to worry about reproving any clauses learned by the SAT solver under the witness ω. It turns out that for all non-trivial proof obligations, the solution-improving constraint $O \geq O\restriction_\alpha$ for the latest solution α obtained is helpful. This also makes it easier to see why the entire pipeline is consistent. During coarse convergence, we never derive $T = 0$, but instead derive $z_k = 0$ for certain values of k using the fact that we could set $T = 0$ wlog. This constraint $z_k = 0$ will be used by the solver for deriving several consequences. Later, when we make the wlog argument that $T \geq s - 1$ for some value s, this incurs the obligation to reprove that $z_k = 0$ holds! That is, the proof checker realizes that $z_k = 0$ was also derived wlog, and we need to prove that this is still consistent with the current wlog assumption to justify that we can "change our mind" about the value of T.

The use of *strengthening-to-core* requires some extra care when dealing with constraint deletions. SAT solvers use heuristics to aggressively erase clauses that are believed to no longer be useful, and this is crucial for performance. Also, clauses in the input are removed whenever some literal in the clause is deduced to be true. In strengthening-to-core mode, we can still do unrestricted deletions of constraints in the derived set $\mathcal{D}$, but a core constraint $C \in \mathcal{C}$ can only be erased if the implication $\mathcal{C} \setminus \{C\} \models C$ can be shown to hold. For this reason we did not implement deletion from the core set in our proof logging routines.

4.3 Stratification

For proof logging of stratification steps as in Section 3.4, we need to be able to convert known facts about the whole objective O to statements about the split objectives O_H and O_L. To certify a unit constraint added during coarse convergence or to derive the constraints $T \geq s - 1$ during fine convergence when maximizing O_H, we need to derive $O_H \geq O_H\restriction_\alpha$ from $O \geq O\restriction_\alpha + 1$. We do this by weakening away all terms in O_L—meaning that for every term $w_i\ell_i$ in O_L we add $w_i\overline{\ell}_i \geq 0$ to cancel the term—to get $O_H \geq O\restriction_\alpha + 1 - g$, where g is the greatest common divisor of the coefficients in O_H. This clearly also entails $O_H \geq O_H\restriction_\alpha - g + 1$. Dividing by g and rounding up yields $\frac{1}{g}O_H \geq \frac{O_H\restriction_\alpha}{g} - 1 + 1$, and multiplying this again by g yields $O_H \geq O_H\restriction_\alpha$.

By applying this reasoning, we can derive the constraint $O_H \geq o^*_H$ right after finding the optimal value o^*_H for O_H. Moreover, after introducing a shadow circuit for $T = s$, we can derive (local) optimality in the form of the constraint $O_H \leq o^*_H$. Hence, we can reformulate the objective by replacing O_H with the constant o^*_H, from which we can now derive the constraint $O_L + o^*_H \geq O\restriction_\alpha +1$. Observe that this constraint coincides with the solution-improving constraint for O_L. Once the constraints $O_L \geq o^*_L$ and $O_L \leq o^*_L$ have been derived in a similar way, the objective will be rewritten to a constant, for which proving optimality boils down to logging a solution that has objective value $o^* = o^*_H + o^*_L$.

4.4 Limiting the Use of Shadow Circuits

Our proof logging method makes repeated use of shadow circuits, which are copies of the original circuit, and repeatedly deriving all constraints defining such circuits could potentially incur serious overhead for proof generation in the solver. Let us discuss ways of limiting or completely eliminating the use of shadow circuits and the downside of such approaches.

First, the shadow circuits are introduced each time the solver deduces a unit clause over an output variable z_k or tare variable t_i. Instead of learning these unit clauses, we could do all subsequent solver calls with those literals as assumptions. At the very end of the fine convergence phase, we could then introduce a single shadow circuit to prove optimality (or, in case of stratification, two shadow circuits: one to prove optimality and one to fix the value of the tare variables). The disadvantage is that when variables used as assumptions, the solver cannot use them to simplify its clause database; so while this would have a positive effect on the time required to do the actual proof logging, it could have negative effects on solving time. Appendix C.2 reports on an experimental evaluation of this approach.

Second, there is a way to completely eliminate shadow circuits. By the end of the execution, the solver knows which value $T = s$ resulted in the final *UNSAT* call in the fine convergence. What we could do at this point is insert at the *beginning* of the proof constraints saying that $T = s$ holds (which at this point can easily be derived by redundance-based strengthening). The rest of the proof will then be checked for a fixed value of T that happens to be the value needed at the end. There are two important reasons why we prefer the shadow circuit approach. The first reason is that it is not clear if and how this would work together with stratification, where after a stratification level we want to fix $T = s - 1$. The second reason is that fixing T in advance adds substantial new information that the solver did not have available when constructing the proof. This means that we would not be verifying that the reasoning the solver actually performed was correct, but only that its reasoning checks out given advance information about the optimal solution. While this could still be used to certify the correctness of the final answer, it would not provide any guarantees about the process leading there. It has been shown repeatedly that proof logging can catch subtle bugs in solvers that only report correct results but for the wrong reasons [EG23, GMM+20, KM21, BBN+23], but in order for this to be possible the correctness of solver-generated proofs should only depend on what the solver actually knows when the proof is being produced.

4.5 Discussion of an Even Simpler Approach and Why It Does Not Work

The proof logging techniques in this paper certify every single reasoning step in the solver. An alternative, and seemingly much simpler, way to get proofs of correctness for *any* MaxSAT solver would be to (i) compute an optimal solution by running the MaxSAT solver without proof logging, (ii) check that this solution is feasible, (iii) encode a solution-improving constraint into CNF, and (iv) call a SAT solver to generate a proof of unsatisfiability (and hence of optimality of the solution) with standard SAT proof logging. However, there are several serious issues with this approach that we would like to point out.

First, proofs of correctness are needed for the CNF encodings used in step (iii), and such proofs cannot be done with SAT proof logging since it cannot reason about values of objective functions. Second, it is not possible to just repeat the "final UNSAT call" of the MaxSAT solver in step (iv). Even if the same SAT solver is used, in the original UNSAT call this solver had access to all constraints learned in previous calls, and there is no guarantee that the solver will learn these constraints again, or other equally good constraints, when it is now run in a different way and with a different input. It is therefore impossible to know for sure whether the final SAT solver invocation with the solution-improving constraint would be faster or, more likely, slower, than the original solving process, and by how much. This defeats the whole idea of generating proofs with a small and predictable overhead, since there would be no way of knowing in advance whether "proof logging" for a previously claimed result would succeed or not. Moreover, when a solution-improving MaxSAT solver makes use of stratification (as discussed in Section 3.4), then optimality is not derived by a single UNSAT call but by a combination of UNSAT calls at different levels. It is hard to see how such a combination of calls could be replicated with the simple approach described above.

Third, an increasingly popular usage scenario for MaxSAT solvers is so-called anytime solving, where the solver can be terminated at any point and then returns the best upper and lower bounds on the objective computed so far. Proofs constructed as described in this paper (as well as in other MaxSAT papers using VeriPB proof logging) will at all times contain formal proofs of everything the solver knows about upper and lower bounds on the objective. Whenever the solver is terminated, it can therefore just end the generated proof at that point by printing a concluding line stating what upper and lower bounds have been proven. This functionality would be lost in the alternative approach.

Finally, even if this approach could be made to work efficiently—which, as explained above, is not really the case, for several reasons—we would have the same problem as in Section 4.4 that we would only certify the final result and not the solver reasoning process.

5 Experimental Evaluation

To evaluate our proof logging approach in practice, we implemented it in the state-of-the-art solution-improving MaxSAT solver Pacose [PRB18]. The source code for all software tools used, as well as all experimental data, are available in [BBN+24]. During development, we extensively checked the correctness of our implementation with a fuzzer [PB23] and minimized failed instances with a delta debugger. This process accelerated the development, as we did not need to create instances for special cases, and helped us fix unexpected and sporadic bugs. The proofs emitted by Pacose were verified by the pseudo-Boolean proof checker VeriPB [Ver], and our fuzzing also helped to debug the proof checker.

The experiments were performed on identical 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 machine and the memory limit was set to 14 GB. The time limits were set to 3 600 seconds for solving a MaxSAT instance with Pacose and to 36 000 seconds for checking the proof with VeriPB. As our benchmark set we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [Max23].

Our implementation supports all techniques Pacose employed in the MaxSAT Evaluation 2023. This means that in addition to the dynamic polynomial watchdog encoding we also implemented proof logging for the binary adder encoding [War98] following the approach in [GMNO22, Van23] as well as support for stratification as described in Section 3.4 and for the preprocessing techniques in TrimMaxSAT [PRB21]. Appendix B discusses TrimMaxSAT in detail and Appendix C contains detailed experimental results for the default setup in which Pacose employs heuristics to choose between different encodings. In this section, we focus on the main novelty of this paper, namely proof logging for SIS with the DPW encoding.

To show the viability of enabling proof logging while solving, we analyse the overhead of generating proofs. In Figure 2 we compare the running time of Pacose with and without proof logging. With proof logging enabled 674 instances were solved within the resource limits, which is 11 fewer instances than without proof logging. Out of the 11 instances that were not solved with proof logging enabled, 9 instances failed due to the memory limit and 2 instances due to the time limit. For the solved instances, Pacose with proof logging was on average 1.93× slower

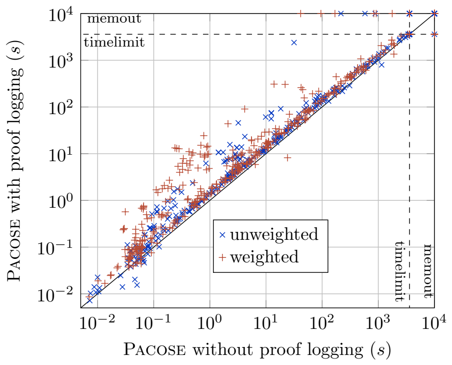


Figure 2: *Proof logging overhead for PACOSE using the DPW encoding.*

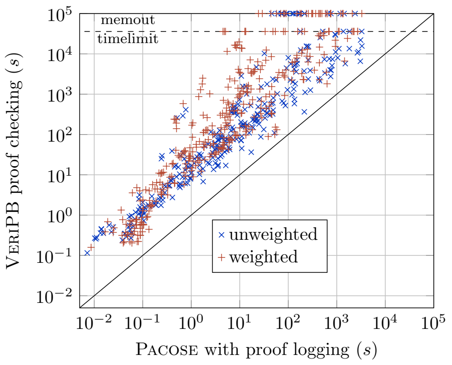


Figure 3: *PACOSE vs. VERIPB running time using DPW encoding.*

than without proof logging. About 90% of the solved instances were solved at most 5.26× more slowly with proof logging enabled. This overhead for solving is to some extent caused by our shadow circuits approach. While we demonstrate that shadow circuits can be used to justify the without loss of generality reasoning in PACOSE, it remains to investigate whether there is a better approach. It is important to note, though, that the average overhead of 1.93× is heavily biased by small instances: the cumulative solving time of all 674 instances, with proof logging is only 1.32× the cumulative solving time without proof logging. This suggests that proof logging overhead decreases for harder instances.

For proof logging to be maximally useful in practice, it is also desirable that it should be possible to check generated proofs within a time limit that is some small constant factor of the solving time for the instance. To evaluate the efficiency of proof checking, we compared the running time of PACOSE with proof logging enabled with the running time of VERIPB, with results plotted in Figure 3. Out of the 674 instances solved by PACOSE with proof logging, 592 were successfully checked by VERIPB, but 53 instances failed due to the memory limit and 29 instances due to the time limit. On average, checking the proof with VERIPB was 22.5× slower than solving and generating the proof with PACOSE. 90% of the proofs were checked within 100× the running time of PACOSE. These results for checking are in line with what has been reported in other works on proof logging for MaxSAT [BBN+23, Van23]. While there is certainly room for further improvements, this shows that proof logging and checking is viable. It should also be emphasized that the only sources of problems for VERIPB were the time and memory limits—other than that all proofs were successfully checked.

6 Conclusion

In this paper, we demonstrate how to design proof logging for solution-improving MaxSAT solving using the dynamic polynomial watchdog encoding. This turns out to be surprisingly challenging, mainly due to the heavy use of reasoning without loss of generality. To understand the correctness of this reasoning at a human level is one thing, but convincing a proof checker by producing machine-verifiable proofs is quite another. What we show is that by combining the redundance-based strengthening rule and the strengthening-to-core mode in VeriPB, together with a technique we call shadow circuits for having more expressive witnessing capabilities, we are able to devise efficient pseudo-Boolean proof logging techniques.

We have implemented our approach in the state-of-the-art MaxSAT solver Pacose. Our experimental evaluation shows that while enabling proof logging is feasible, it does incur a non-negligible overhead in solving time. Moreover, the time needed to check the generated proofs is several times larger than the time needed to generate them, suggesting that more efficient algorithms and more optimized engineering are needed in VeriPB. This is not so surprising, since the focus of VeriPB development so far has been on providing support for certifying algorithms in combinatorial optimization paradigms previously beyond the reach of proof logging, rather than on optimizing the proof checker code base.

The addition of Pacose to the collection of certifying MaxSAT solvers using VeriPB proofs provides further support to the hypothesis that pseudo-Boolean proof logging hits a sweet spot for MaxSAT solving, being rich enough to support a wide variety of solving algorithms and complex reasoning tricks, but still being simple enough to support even formally verified proof checking as in [BMM+23, GMM+24, IOT+24].

We believe that in the longer term VeriPB can have a strong positive impact on the reliability and robustness of MaxSAT solvers. In the other direction, MaxSAT solving is likely to provide excellent benchmarks and performance challenges to further improve pseudo-Boolean proof logging and checking. Our suggestion for speeding up these developments is to introduce a certifying track in the yearly MaxSAT Evaluation [Max].

Acknowledgements

We want to thank Florian Pollitt and Mathias Fleury for their assistance with the CaDiCaL proof tracer and for fuzzing VeriPB within CaDiCaL. Their contributions were very helpful to further improve the robustness of the VeriPB toolchain. We also wish to acknowledge useful discussions with participants of the Dagstuhl workshop 23261 *SAT Encodings and Beyond*. The computational experiments were enabled by resources provided by LUNARC at Lund University.

This work has been financially supported by the Research Council of Finland under grants 342145, the Swedish Research Council grant 2016-00782, the Independent Research Fund Denmark grant 9040-00389B, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and the Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N).

Appendix A Formalization of the Proof Logging of SIS with the DPW

In this appendix, we provide formal details on the claims made in the main body of the paper. In the proofs, we follow the same notation. The formalization of the reasoning in the coarse convergence is discussed in Section 4.2, here we discuss the other phases.

A.1 Coarse Convergence

Our first proposition formalizes the wlog performed during the coarse convergence phase.

Proposition 3 (Proposition 2, restated)**.** *Assume the definition of z_k has been derived and a complete shadow circuit for $T = 0$ has been introduced. Furthermore assume the constraint*

$$O \geq 1 + k \cdot 2^P \tag{4}$$

has been derived. The constraint $\overline{z}_k \geq 1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto 0, Y \mapsto Y^{T=0}.$$

The notation for the witness in this proposition is a shorthand for the mapping that sends each variable t_i to 0 and every introduced circuit variable y to the corresponding shadow circuit variable $y^{T=0}$.

Proof. To verify this is indeed possible, we need to show that from

$$\mathcal{C} \cup \mathcal{D} \cup \{z_k \geq 1\}$$

we can derive the following constraints:

- $\overline{z}_k\restriction_\omega \geq 1$; in other words we need to show that $\overline{z}_k^{T=0} \geq 1$ holds. Recall that $z_k^{T=0}$ is defined by the reification

  $$\overline{z}_k^{T=0} \Leftrightarrow O - 0 \geq 1 + k \cdot 2^P.$$

  Adding up one direction of this definition to (4), immediately yields that $\overline{z}_k^{T=0} \geq 1$, as desired.

- $C\restriction_\omega$ for each $C \in \mathcal{C}$.
    - If C is a clause in the original input, $C\restriction_\omega = C$ and this is trivial.
    - If C is a previously derived solution-improving constraint, also $C\restriction_\omega = C$ (since ω does not touch any variable in O.
    - If C is a previously derived constraint of the form $\overline{z}_{k'} \geq 1$ with $k' < k$, this can either be derived analogously to $\overline{z}_k\restriction_\omega \geq 1$ or directly from the fact that the definitions of z_k and z'_k immediately imply that $z_k = 0$ implies that $z_{k'} = 0$.
- $O\restriction_\omega \geq O$; this is obvious since the variables in O are unaltered by ω. □

Remark 1. Proposition 3 assumes the existence of a constraint (4). It can be seen that this constraint is actually a (potentially weakened version of a) non-strict solution improving constraint $O \geq O\restriction_\alpha$ where α is a previously found solution. During the coarse convergence phase, this constraint can be obtained by weakening the solution-improving constraint.

At the end of the coarse convergence phase, also the unit clause $z_{k^*} \geq 1$ is derived. This requires no additional proof logging: this clause is obtained by running the SAT solver with the assumption that $z_{k^*} = 0$ and failing. Whenever this is the case; we know that $z_{k^*} \geq 1$ is internally derived by standard conflict analysis; hence this constraint is added to $\mathcal{D}$ without any additional effort.

A.2 Fine Convergence

As with the coarse convergence, the constraints derived during fine convergence that require a justification in the proof are the unit clauses added to the solver. Proving this relies again on redundance-based strengthening and a shadow circuit.

Proposition 4. *Assume $\overline{z}_{k^*-1} \geq 1$ has been derived. Let s be any number and assume a complete shadow circuit for $T = s - 1$ has been introduced. Furthermore assume the constraint*

$$O \geq s + (k^* - 1) \cdot 2^P \tag{5}$$

has been derived. The constraint $T \geq s - 1$ can be derived using redundance-based strengthening with witness

$$\omega = T \mapsto s, Y \mapsto Y^{T=s-1}.$$

Proof. As in the proof of Proposition 2, this yields several proof obligations. The only non-trivial ones are

- Previously derived constraints of this form $T \geq s' - 1$, but they are trivially satisfied under ω since $s \geq s'$.

- The unit clause $\overline{z}_{k^*-1} \geq 1\restriction_\omega$. In other words we need to show that $\overline{z}_{k^*-1}^{T=s-1}$ holds. Recall that $z_{k^*-1}^{T=s-1}$ is defined by the reification

  $$\overline{z}_{k^*-1}^{T=s-1} \Leftrightarrow O-(s-1) \geq 1+(k^*-1)\cdot 2^P$$

  which simplifies to

  $$\overline{z}_{k^*-1}^{T=s-1} \Leftrightarrow O-s \geq (k^*-1)\cdot 2^P.$$

  Now (5) tells us precisely that the right-hand side of this equivalence is satisfied, hence a straightforward cutting planes derivation indeed allows us to conclude that $\overline{z}_{k^*-1}^{T=s} \geq 1$. □

Remark 2. Just like Proposition 2, also Proposition 4 does not make use of the model-improving constraint, but rather makes the assumption on O it uses explicit in (5). As before, this turns out to be useful when applying Proposition 4 in the context of stratification.

Proposition 4 will be applied when a solution α is found taking

$$s := O\restriction_\alpha - (k^*-1)\cdot 2^P.$$

In this case, the solution-improving tells us that

$$O \geq O\restriction_\alpha +1 = s+(k^*-1)\cdot 2^P+1,$$

and (5) is indeed satisfied. Unit clauses are derived if for a certain j, $s \geq 2^P - 2^j + 1$. In this case, the derived constraint $T \geq s-1$ guarantees that $T \geq 2^P - 2^j$, i.e., that all dominant bits of T up to j must be equal to one. This follows using reverse unit propagation or a straightforward cutting planes derivation.

A.3 Conclusion of Optimality

When the very last call to the SAT solver is unsatisfiable, we need to derive a contradiction in the proof, to complete the proof that the previously best found solution is optimal. We proceed as follows. First, we introduce a fresh variable, let us call it p using the reification

$$p \Leftrightarrow O \geq o^* + 1. \tag{6}$$

Our goal will be to show that p is false, which then allows us to conclude that the objective can no longer be improved, meaning we have indeed proven optimality. Recall that at this point, we have s defined as $s := o^* - (k^*-1)\cdot 2^P$. The crucial step in our proof is showing that without loss of generality T can be set equal to s. We proceed as follows.

Proposition 5. *Assume $\overline{z}_{k^*-1} \geq 1$ and the definition of p have been derived. Furthermore suppose that a shadow circuit for $T = s$ has been introduced. Using redundance-based strengthening with witness*

$$\omega = T \mapsto s, Y \mapsto Y^{T=s}$$

we can derive the PB constraints representing

$$p \Rightarrow T = s, \tag{7}$$

i.e., in normalised form, the constraints

$$s \cdot \overline{p} + T \geq s, \text{ and} \tag{8}$$

$$(2^P - s - 2) \cdot \overline{p} + \sum_{j=0}^{P-1} 2^j \cdot \overline{T}_j \geq (2^P - 1) - s - 1. \tag{9}$$

Proof. The proof for the two constraints is similar. The only proof goal where they differ is showing that the constraint to-be-derived is satisfied under ω, but this is trivial since the witness sets T equal to s by construction.

For all the other proof goals, we can make use the negation of the constraint to be derived (the negation of (8) or of (9)). From this negation, we can directly derive $p \geq 1$. Adding this up to (one direction of (6) yields $O \geq o^* + 1$, i.e., that

$$O \geq s + (k^* - 1) \cdot 2^P + 1. \tag{10}$$

In other words, the conditions of 4 are satisfied. All the other proof obligations are the same as the ones in the proof of that proposition and hence, making use of (10), the proof proceeds identically to the proof of Proposition 4. □

In words, Proposition 5 tells us is that *if* the objective is strictly improving on the previously found best value, *then* we can set T equal to s without loss of generality. The SAT solver, however, has in its last call that yielded UNSAT already derived a clause telling us that at least one of the bits of T does not correspond to s. So we can now straightforwardly derive that $\overline{p} \geq 1$ and hence that $O \leq o^*$, which is what we needed for concluding optimality.

Appendix B Proof Logging of Additional Techniques Implemented in Pacose

We detail some of the additional search techniques implemented in and how we proof log them. As a minor point, we note for completeness that in addition to the gcd-based criterion described in Section 3.4, Pacose attempts to find more partitions of the objective during stratification via exhaustive search, as illustrated by the following example:

Example 3. Consider the objective $O := 14x_1 + 9x_2 + 5x_3 + 2x_4 + 1x_5 + 1x_6$ and the partition $H = \{1, 2, 3\}$ and $L = \{4, 5, 6\}$. According to the gcd-based criterion from Section 3.4, this partition is not viable due to the gcd not aligning with any single divisor that groups the weights cohesively. However, this partition still validly separates the weights of x_1 to x_6 through an alternative method: Define L_C as the set containing all possible summed combinations of weights from L: $L_C := 5, 9, 14, 5 + 9, 5 + 14, 9 + 14, 5 + 9 + 14$. To validate this partitioning, ensure that the total weight W_L from L is at most the difference between any two sums in L_C. This ensures that L forms a consistent grouping, as there is no weight combination of L invalidating a prior result of solving H.

A more in-depth explanation together with a proof can be found in [PRB21]. While certifying the exhaustive search remains interesting future work, we note that it did not result in additional partitions on any of the benchmarks in our evaluation, nor on the weighted instances of the 2019 and 2020 MaxSAT Evaluation.

We would like to mention that a naive approach to certify the exhaustive search would be to derive the desired constraint $O_H \geq O_H\restriction_\alpha$ from the weakened constraint $O_H \geq O\restriction_\alpha - W_L + 1$ using redundance-based strengthening with an empty witness. As $O_H\restriction_\alpha$ is the sum of a subset of the coefficients in O_H and the distance between any two sums is at least W_L, the negation $O_H < O_H\restriction_\alpha$ of the desired constraint can only be satisfied if the sum of true literals in O_H is at most $O_H\restriction_\alpha - W_L$. As $O\restriction_\alpha \geq O_H\restriction_\alpha$, the weakened constraint can only be satisfied if the sum of true literals in O_H is at least $O_H\restriction_\alpha - W_L + 1$. Hence, there exists no assignment to the variables in O_H for which both constraints are satisfied. To show this we can iterate through every possible assignment α of the variables in O_H and derive the clause excluding this assignment by reverse unit propagation. This step works, as reverse unit propagation for this clause assigns all variables in O_H, which will falsify either the negated constraint or the weakened constraint by the arguments above. Resolving all the clauses will result in a contradiction that proves that $O_H \geq O_H\restriction_\alpha$ is implied.

B.1 TrimMaxSAT

TrimMaxSAT [PRB21] is a preprocessing technique applied before the main SIS algorithm in order to decrease the number of literals in the objective that need to be encoded by the DPW and to get a good initial value of the objective. TrimMaxSAT heuristically splits the variables in the objective into partitions and queries the SAT solver for a solution that assigns at least one of the literals in each partition to 1. If such an assignment is found, the objective variables set to 1 are removed from consideration and the number of partitions are decreased. If the partition size is 1 and the SAT solver reports UNSAT, all remaining literals are fixed to 0 for the rest of the search. In other words TrimMaxSAT aims to find objective literals whose negation is implied by the constraints in the formula and fix their value, thus conceptually decreasing the size of the objective under consideration and–as a consequence–also the size of the DPW encoding built over it.

In more detail, assume $\mathcal{L}$ contains the set of objective variables that have not been set to 1 in any solutions found so far during TrimMaxSAT. During an iteration of TrimMaxSAT, $\mathcal{L}$ is partitioned into m subsets $\mathcal{L}^i$ for $i = 1, \ldots, m$. A new variable r is introduced and the clauses $r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ for every $i = 1, \ldots, m$ are added to the SAT solver and the proof via redundance-based strengthening to the core set. The SAT solver is then queried under the assumption that r is true. If the result is SAT, the literals in $\mathcal{L}$ assigned to 1 in the obtained solution are removed from the set under consideration and the unit clause $\overline{r} \geq 1$ is added to the solver such that the SAT solver can remove the clauses of the form $r \Rightarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$. This unit clause can be derived by redundance-based strengthening with witness $\omega = r \mapsto 0$. If, on the other hand, the result is UNSAT, the unit clause $\overline{r} \geq 1$ is added to the SAT solver and the SAT solver can simplify its clause database. This clause is derived by standard cutting planes reasoning in the conflict analysis by the SAT solver and is therefore added to the derived set in the proof. If in this case $m = 1$, we can also conclude that all literals $\ell \in \mathcal{L}$ are implied to be false. Hence, the solver learns the unit clauses $\overline{\ell} \geq 1$. In order to derive $\overline{\ell} \geq 1$ for each $\ell \in \mathcal{L}^i$, we first introduce the second part of the reification $r \Leftarrow (\sum_{\ell \in \mathcal{L}^i} \ell \geq 1)$ using the redundance rule with witness $r \mapsto 1$ and then use cutting planes reasoning to derive that since r is false, all literals in $\mathcal{L}^i$ must be false. Interestingly, thanks to the use of strengthening-to-core, the unit clause $\overline{r} \geq 1$ derived earlier does not interfere with the derivation of the second direction of the reification.

B.2 Hardening

Hardening refers to the addition of the unit clause l_i for an objective literal l_i if the currently best known solution o^* is larger than the sum of all weights in O excluding w_i. In the proof, the unit clause l_i can be derived easily from the solution-improving constraint and the objective reformulation rule can be used to replace l_i by the constant w_i in the objective.

Appendix C Additional Experimental Evaluation

In this appendix, we present some additional experimental analysis with data and plots to give some further insights into proof logging for Pacose. In Section C.1, we present results for the binary adder encoding that is also used in Pacose and how detail how well proof logging performs for Pacose when it heuristically selects the encoding. We present data for an additional approach that uses assumptions instead of unit clauses for fixing variables in the coarse convergence in Section C.2. To better understand the proof logging overhead in Pacose, we have a deeper look at some additional data for the proof logging process in Section C.3.

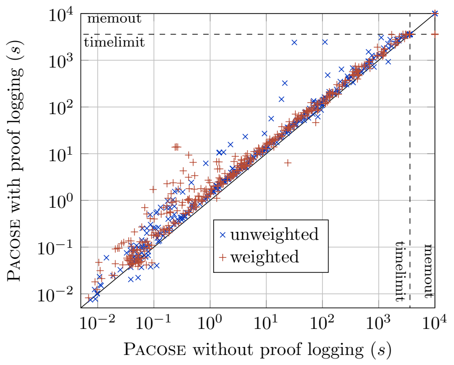


Figure 4: *Proof logging overhead for Pacose using the binary adder encoding.*

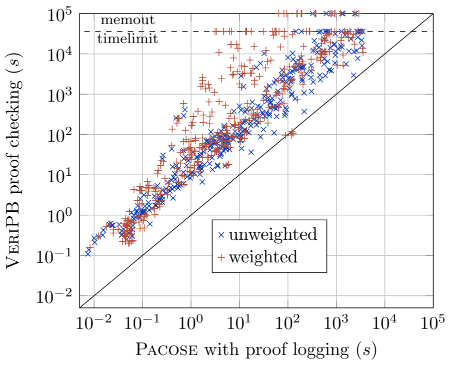


Figure 5: *Pacose vs. VeriPB running time using binary adder encoding.*

C.1 Binary Adder Encoding and Encoding Selection Heuristic

Pacose also uses the binary adder encoding [War98] instead of the DPW encoding. A comparison between these two encodings is beyond the scope of this paper, but as we implemented proof logging for both encodings, we can also have a look at the data for the binary adder encoding. A comparison of solving with and without proof logging for this encoding can be found in Figure 4. With proof logging for the binary adder encoding 722 instances could be solved within the resource limits, which are 6 fewer instances than without proof logging. This also demonstrates that the heuristic for selecting the encoding works, as the number of solved instances for the heuristic is bigger than for any of the two encodings on their own. In the mean, Pacose with proof logging is 1.63× slower than without proof logging. This overhead is smaller than for the DPW encoding, which lead to the conclusion that more work is required to certify the DPW encoding compared to the binary adder encoding.

Out of the 722 instances that were solved with the binary adder encoding, 658 instances were successfully checked by VeriPB within the resource limits. In Figure 5, the running time of Pacose is compared to that of VeriPB. In the mean, VeriPB is 21.1× slower than Pacose for solving the instance with proof logging, which is similar to the DPW encoding. This could mean that the bottleneck for checking the proofs is the implementation of the checker.

Using the default settings, Pacose heuristically selects between the DPW and binary adder encoding. A plot comparing Pacose with and without proof logging in the default settings in Figure 6 and a plot comparing Pacose with proof logging with VeriPB for checking the proof in Figure 7. With this heuristic activated, 698 instances are solved within the resource limits with proof logging enabled and 707 instances without. Pacose with proof logging is 1.83× slower in the mean than

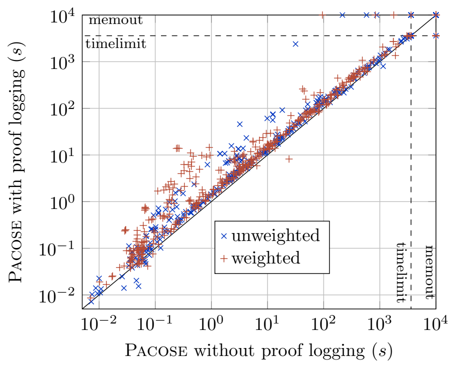


Figure 6: *Proof logging overhead for* PACOSE *using heuristic encoding selection.*

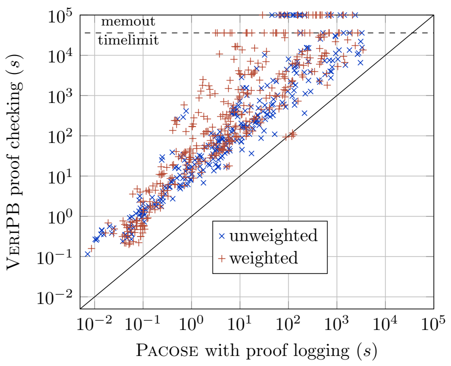


Figure 7: PACOSE *vs.* VERIPB *running time using heuristic encoding selection.*

PACOSE without proof logging. Checking the proof with VERIPB is 21.8× slower than running PACOSE with proof logging in the mean.

C.2 Coarse Convergence with Assumptions Instead of Unit Clauses

An alternative approach for representing the information that output variables of the DPW encoding are fixed to a value in the coarse convergence is to use additional assumptions for the SAT solver instead of unit clauses. As we need a shadow circuit to derive each unit clause, we could reduce the number of shadow circuits by using assumptions. The idea is that we add the variable fixing to the assumptions for all future calls to the SAT solver. This approach is supported in PACOSE, and we ran additional experiments using this approach.

The following data always use assumptions instead of unit clauses for fixing variables. In Figure 8, PACOSE with proof logging is compared to PACOSE without proof logging. Using assumptions PACOSE with proof logging could solve 666 instances, which is 10 fewer instances than without proof logging. PACOSE with proof logging is 1.81× slower than without proof logging in the mean. This is very similar to PACOSE with the DPW encoding where the variables are fixed by unit clauses and introducing shadow circuits. In the mean, the proof checking is 22.2× slower than solving the instance with proof logging.

It can be concluded that this alternative approach of fixing variables by adding assumptions is about as good as doing the fixing by unit clauses. Hence, it could be that introducing additional shadow circuits for deriving the unit clauses does not slow down the solving a lot, or it is a coincidence that the performance gains are countered by the additional work required for keeping track of the assumptions.

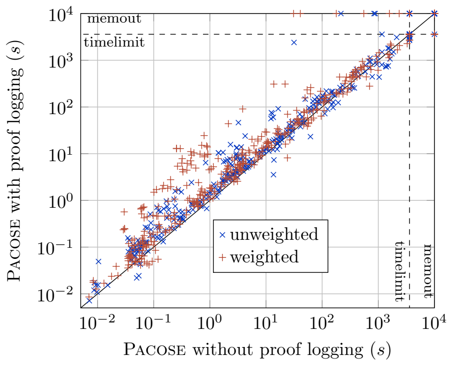


Figure 8: *Proof logging overhead for Pacose using DPW encoding and assumptions.*

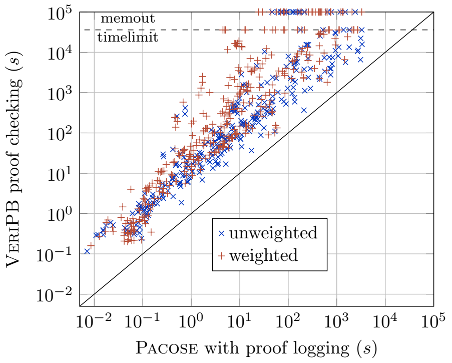


Figure 9: *Pacose vs. VeriPB running time using DPW encoding and assumptions.*

C.3 Proof Logging Overhead Analysis

To get a better understanding of the 1.93× slowdown of Pacose with proof logging compared to without proof logging, we investigate different causes for the extra running time with proof logging. The idea for doing so is to get insights into how to improve the running time of the solvers.

The expectation is that the proof size scales linearly with the running time of the solver. It would be interesting to look into the instances where this is not the case and if there is a correlation with the solving overhead. We can illustrate this by plotting the solving time against the proof size and colour the marks depending on the overhead as it is done in Figure 10 for the DPW encoding and in Figure 11 for the binary adder encoding. We added a diagonal line representing linear scaling of proof size with running time for better orientation, which is not related to the data at all. It can be seen that for the instances that have a proof size that is significantly bigger than expected, the overhead also seems to increase similarly. To confirm this observation, we compute the correlation of the proof logging overhead and the proof size divided by the solving time. For the DPW encoding we have a correlation of 0.92 and for the binary adder encoding we have a correlation of 0.88, which shows that the two parameters are highly correlated. This mean that the slowdown is due to proof being larger than expected for some instances.

We can conclude with some ideas to improve the performance of proof logging in Pacose. First, the performance can be improved by engineering better data structures to handle the proof logging to increase the speed for writing the proof. This idea only works if we have not reached the maximum persistent disk write speed, which is not the case for our experiments. Second, the proof could be done in a smarter way to reduce the size of the proof, where slow parts of the proof logging could be identified by profiling. Considering that we also have a 1.63×

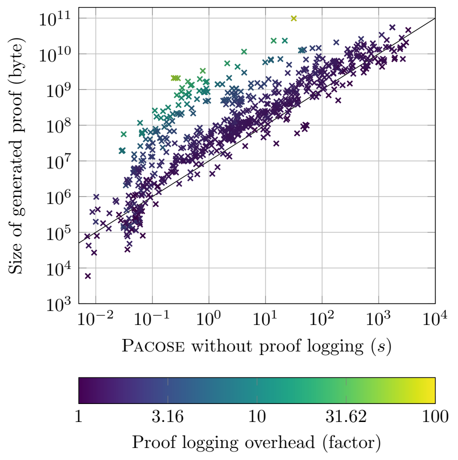


Figure 10: *Solving time vs. proof size vs. solving overhead for proof logging for the DPW encoding.*

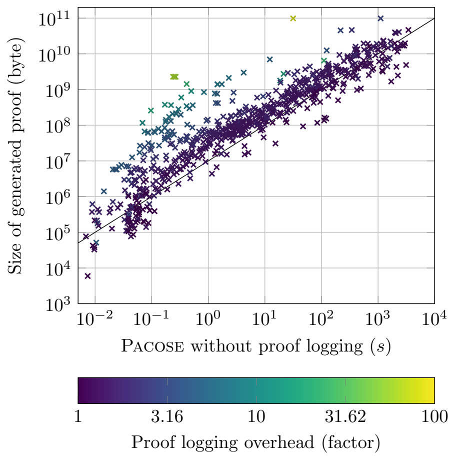


Figure 11: *Solving time vs. proof size vs. solving overhead for proof logging for the binary adder encoding.*

slowdown for the binary adder encoding, the slowdown is not purely caused by the shadow circuits, as they are not used for this encoding.

References

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[ALM09] Josep Argelich, Inês Lynce, and João P. Marques-Silva. On solving Boolean multilevel optimization problems. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 393–398, July 2009.

[BB03] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP '03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, September 2003.

[BB09] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories (SMT '09)*, pages 1–5, August 2009.

[BBN$^+$23] 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.

[BBN$^+$24] Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, Tobias Paxian, and Dieter Vandesande. Experimental Repository for "Certifying Without Loss of Generality Reasoning in Solution-Improving Maximum Satisfiability", June 2024.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, June 2009.

[BCH21] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. A flexible proof format for SAT solver-elaborator communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '21)*, volume 12651 of *Lecture Notes in Computer Science*, pages 59–75. Springer, March-April 2021.

[BGMN23] 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*.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJM21] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Biere et al. [BHvMW21], chapter 24, pages 929–991.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, June 2007. Extended version of paper in *SAT '06*.

[BMM$^+$23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BRK$^+$22] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength SMT solver. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 15–35. Springer, August 2022.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[EG23] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 197(2):793–812, February 2023.

[EH20] Salomé Eriksson and Malte Helmert. Certified unsolvability for SAT planning with property directed reachability. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling*, pages 90–100, October 2020.

[ERH17] Salomé Eriksson, Gabriele Röger, and Malte Helmert. Unsolvability certificates for classical planning. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS '17)*, pages 88–97, June 2017.

[ERH18] Salomé Eriksson, Gabriele Röger, and Malte Helmert. A proof system for unsolvable planning tasks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS '18)*, pages 65–73, June 2018.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In *Proceedings of the 1st International Workshop on Bounded Model Checking (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560, July 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[Fle20] Mathias Fleury. *Formalization of Logical Calculi in Isabelle/HOL*. PhD thesis, Universität des Saarlandes, 2020. Available at `https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28722`.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. 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 252–265. Springer, August 2006.

[GMM$^+$20] 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.

[GMM$^+$24] 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 368h AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GMNO22] 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.

[GN03] 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.

[GN21] 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.

[GSD19] 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.

[HOGN24] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

[IOT$^+$24] 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.

[JHB12] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, June 2012.

[JMM15] Saurabh Joshi, Ruben Martins, and Vasco M. Manquinho. Generalized totalizer encoding for pseudo-Boolean constraints. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP '15)*, volume 9255 of *Lecture Notes in Computer Science*, pages 200–209. Springer, August-September 2015.

[KM21] Sonja Kraiczy and Ciaran McCreesh. 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)*, pages 1396–1402, August 2021.

[KP19] Michal Karpinski and Marek Piotrów. Encoding cardinality constraints using multiway merge selection networks. *Constraints*, 24(3–4):234–251, October 2019.

[LBJ20] Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete MaxSAT solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, volume 325 of *Frontiers* *in Artificial Intelligence and Applications*, pages 347–354, August-September 2020.

[LM21] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In Biere et al. [BHvMW21], chapter 23, pages 903–927.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, January 2011.

[Max] MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. `https://maxsat-evaluations.github.io/`.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MM11] António Morgado and João P. Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pages 924–926, November 2011.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MPS14] Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of pseudo-Boolean constraints into CNF such that generalized arc consistency is maintained. In *Proceedings of the 37th Annual German Conference on Artificial Intelligence (KI '14)*, volume 8736 of *Lecture Notes in Computer Science*, pages 123–134. Springer, September 2014.

[PB23] Tobias Paxian and Armin Biere. Uncovering and classifying bugs in MaxSAT solvers through fuzzing and delta debugging. In *Proceedings of the 14th International Workshop on Pragmatics of SAT*, volume 3545 of *CEUR Workshop Proceedings*, pages 59–71. CEUR-WS.org, July 2023.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. 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)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] 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.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, January 2021.

[Rög17] Gabriele Röger. Towards certified unsolvability in classical planning. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI '17)*, pages 5141–5145, August 2017.

[SFBF21] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving (PxTP '21*, volume 336 of *Electronic Proceedings in Theoretical Computer Science*, pages 49–54, July 2021.

[Sin05] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP '05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, October 2005.

[Van23] Dieter Vandesande. Towards certified MaxSAT solving: Certified MaxSAT solving with SAT oracles and encodings of pseudo-Boolean constraints. Master's thesis, Vrije Universiteit Brussel (VUB), 2023.

[VDB22] 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.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

[War98] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, October 1998.

[WHH14] 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.

Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs

Abstract

It is well known that reformulating the original problem can be crucial for the performance of mixed-integer programming (MIP) solvers. To ensure correctness, all transformations must preserve the feasibility status and optimal value of the problem, but there is currently no established methodology to express and verify the equivalence of two mixed-integer programs. In this work, we take a first step in this direction by showing how the correctness of MIP presolve reductions on 0–1 integer linear programs can be certified by using (and suitably extending) the VeriPB tool for pseudo-Boolean proof logging. Our experimental evaluation on both decision and optimization instances demonstrates the computational viability of the approach and leads to suggestions for future revisions of the proof format that will help to reduce the verbosity of the certificates and to accelerate the certification and verification process further.

1 Introduction

Boolean satisfiability solving (SAT) and *mixed-integer programming (MIP)* are two computational paradigms in which surprisingly mature and powerful solvers have been developed over the last decades. Today such solvers are routinely used to solve large-scale problems in practice despite the fact that these problems are *NP*-hard. Both SAT and MIP solvers typically start by trying to simplify the input problem before feeding it to the main solver algorithm, a process known as *presolving* in MIP and *preprocessing* in SAT. This can involve, e.g., fixing

Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. "Certifying MIP-Based Presolve Reductions for 0–1 Integer Linear Programs". In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, volume 14742 of *Lecture Notes in Computer Science*, pages 310–328. Springer, May 2024.

variables to values, strengthening constraints, removing constraints, or adding new constraints to break symmetries. Such techniques are very important for SAT solver performance [BJK21], and for MIP solvers they often play a decisive role in whether a problem instance can be solved or not, regardless of whether the solver uses floating-point [ABG$^+$19] or exact rational arithmetic [EG22].

The impressive performance gains for modern combinatorial solvers come at the price of ever-increasing complexity, which makes these tools very hard to debug. It is well documented that even state-of-the-art solvers in many paradigms, not just SAT and MIP, suffer from errors such as mistakenly claiming infeasibility or optimality, or even returning "solutions" that are infeasible [AGJ$^+$18, CKSW13a, GSD19, Klo14, Ste11]. During the last decade, the SAT community has dealt with this problem in a remarkably successful way by requiring that solvers should use *proof logging*, i.e., produce machine-verifiable certificates of correctness for their computations that can be verified by a stand-alone proof checker. A number of proof formats have been developed, such as DRAT [HHW13a, HHW13b, WHH14], GRIT [CMS17], and LRAT [CHH$^+$17], which are used to certify the whole solving process including preprocessing.

Achieving something similar in a general MIP setting is much more challenging, amongst others because of the presence of continuous and general integer variables, which may even have unbounded domains. For numerically exact MIP solvers [CKSW13b, EG22, EG24] the proof format VIPR [CGS17] has been introduced, but it currently only allows verification of feasibility-based reasoning, which must preserve all feasible solutions. In particular, it does not support the verification of dual presolving techniques that may exclude feasible solutions as long as one optimal solution remains. This means that while exact MIP solvers could in principle generate a certificate for the main solving process, such a certificate would only establish correctness under the assumption that all the presolving steps were valid, as, e.g., in [EG22]. And, unfortunately, the proof logging techniques for SAT preprocessing cannot be used to address this problem, since they can only reason about clausal constraints.

Our contribution in this work is to take a first step towards verification of the full MIP solving process by demonstrating how pseudo-Boolean proof logging with VeriPB can be used to produce certificates of correctness for a wide range of MIP presolving techniques for 0–1 integer linear programs (ILPs). VeriPB is quite a versatile tool in that it has previously been employed for certification of, e.g., advanced SAT solving techniques [BGMN23, GN21], SAT-based optimization (MaxSAT) [BBN$^+$23, VDB22], subgraph solving [GMM$^+$20, GMN20], and constraint programming [GMN22, MM23]. However, to the best of our knowledge this is the first time the tool has been used to prove the correctness of reformulations of optimization problems, and this presents new challenges. In particular, the proof system turns out not to be well suited for problem reformulations with frequent changes to the objective function, and therefore we introduce a new rule for objective function updates.

Our computational experiments confirm that this approach to certifying presolve reductions is computationally viable and the overhead for certification aligns with what is known from the literature for certifying problem transformations in other contexts [GMNO22]. The analysis of the results reveals new insights into performance bottlenecks, and these insights directly translate to possible revisions of the proof logging format that would be valuable to address in order to decrease the size of the generated proofs and speed up proof verification.

We would like to note that, while our current methods are only applicable to 0–1 ILPs, this covers already a large and important class of MIPs. In particular, there are applications where the exact and verified solution of 0–1 ILPs is highly relevant, see [Ach07, EGP22, SBD19] for some examples.

The rest of this paper is organized as follows. After presenting pseudo-Boolean proof logging and VeriPB in Sec. 2, we demonstrate in Sec. 3 how to produce VeriPB certificates for MIP presolving on 0–1 ILPs. In Sec. 4 we report results of an experimental evaluation, and we conclude in Sec. 5 with a summary and discussion of future work.

2 Pseudo-Boolean Proof Logging with VeriPB

We start by reviewing pseudo-Boolean reasoning in Sec. 2.1, and then explain our extension to deal with objective function updates in Sec. 2.2. In order to make the concept of proof logging more concrete, we conclude this section by providing, in Tab. 1, a few examples of how the derivation rules explained below are encoded in VeriPB syntax. For space reasons, this list does not include examples of subproofs that may be necessary for some derivations that cannot be proven automatically by VeriPB. Further details on practical aspects and implementation of pseudo-Boolean proof logging can be found in the software repository of VeriPB [GO23].

2.1 Pseudo-Boolean Reasoning with the Cutting Planes Method

Our treatment of this material will by necessity be somewhat terse—we refer the reader to [BN21] for more information about the cutting planes method and to [BGMN23, GMNO22] for detailed information about the VeriPB proof system and format.

We write x to denote a $\{0, 1\}$-valued variable and $\overline{x}$ as a shorthand for $1 - x$, and write ℓ to denote such *positive* and *negative literals*, respectively. By a *pseudo-Boolean (PB) constraint* we mean a 0–1 linear inequality $\sum_j a_j \ell_j \geq b$, where when convenient we can assume all literals ℓ_j to refer to distinct variables and all a_j and b to be non-negative (so-called *normalized form*). A *pseudo-Boolean formula* is just another name for a 0–1 integer linear program. For optimization problems we also have an objective function $f = \sum_j c_j x_j$ that should be minimized (and f can be negated to represent a maximization problem).

The foundation of VeriPB is the *cutting planes* proof system [CCT87]. At the start of the proof, the set of *core constraints* $\mathcal{C}$ are initialized as the 0–1 linear inequalities in the problem instance. Any constraints derived as described below are placed in the set of *derived constraints* $\mathcal{D}$, from where they can later be moved to $\mathcal{C}$ (but not vice versa). Loosely speaking, VeriPB proofs maintain the invariant that the optimal value of any solution to $\mathcal{C}$ and to the original input problem is the same. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ by performing *addition* of two constraints or *multiplication* of a constraint by a positive integer, and *literal axioms* $\ell \geq 0$ can be used at any time. Additionally, for a constraint $\sum_j a_j \ell_j \geq b$ written in normalized form we can apply *division* by a positive integer d followed by rounding up to obtain $\sum_j \lceil a_j/d \rceil \ell_j \geq \lceil b/d \rceil$, and *saturation* can be applied to yield $\sum_j \min\{a_j, b\} \cdot \ell_j \geq b$.

For a PB constraint $C \doteq \sum_j a_j \ell_j \geq b$ (where we use $\doteq$ to denote syntactic equality), the negation of C is $\neg C \doteq \sum_j a_j \ell_j \leq b - 1$. For a *partial assignment* ρ mapping variables to $\{0, 1\}$, we write $C\!\upharpoonright_\rho$ for the *restricted constraint* obtained by replacing variables in C assigned by ρ by their values and simplifying the result. We say that C *unit propagates* ℓ *under* ρ if $C\!\upharpoonright_\rho$ cannot be satisfied unless ℓ is assigned to 1. If unit propagation on all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$ starting with the empty assignment $\rho = \emptyset$, and extending ρ with new assignments as long as new literals propagate, leads to contradiction in the form of a violated constraint, then we say that C follows by *reverse unit propagation (RUP)* from $\mathcal{C} \cup \mathcal{D}$. Such (efficiently verifiable) RUP steps are allowed in VeriPB proofs when it is convenient to avoid writing out an explicit derivation of C from $\mathcal{C} \cup \mathcal{D}$. We will also write $C\!\upharpoonright_\omega$ to denote the result of applying to C a *(partial) substitution* ω which can remap variables to other literals in addition to 0 and 1, and we extend this notation to sets in the obvious way by taking unions.

In addition to the cutting planes rules, which can only derive semantically implied constraints, VeriPB has a *redundance-based strengthening rule* that can derive a non-implied constraint C as long as this does not change the feasibility or optimal value of the problem. Formally, C can be derived from $\mathcal{C} \cup \mathcal{D}$ using this rule by exhibiting in the proof a *witness substitution* ω together with subproofs

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\})\!\upharpoonright_\omega \cup \{f \geq f\!\upharpoonright_\omega\}, \tag{1}$$

of all constraints on the right-hand side from the premises on the left-hand side using the derivation rules above. Intuitively, what (1) shows is that if α is any assignment that satisfies $\mathcal{C} \cup \mathcal{D}$ but violates C, then $\alpha \circ \omega$ satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and yields at least as good a value for the objective function f.

During presolving, constraints in the input formula can be deleted or replaced by other constraints, and the proof needs to establish that such modifications are correct. While deletions from the derived set $\mathcal{D}$ are always in order, removing a constraint from the core set $\mathcal{C}$ could potentially introduce spurious solutions. Therefore, deleting a constraint C from $\mathcal{C}$ can only be done by the *checked deletion rule*, which requires to show that C could be rederived from $\mathcal{C} \setminus \{C\}$ by redundance-based strengthening (see [BGMN23] for a more detailed explanation).

2.2 A New Rule for Objective Function Updates

When variables are fixed or identified during the presolving process, the objective function f can be modified to a function f'. This modified objective f' can then

Table 1: *Examples of basic derivation rules in VeriPB syntax. Here, (id) refers to the constraint ID assigned by VeriPB.*

Rule	Syntax	Explanation
cutting planes in reverse Polish notation	`pol x1 4 +`	add $x_1 \geq 0$ and (4)
	`pol 3 2 d`	divides (3) by 2
	`pol 1 2 * ~x1 +`	multiplies (1) by 2 and adds $\overline{x}_1 \geq 0$
redundance-based strengthening	`red +1 x1 >= 1; x1 1`	verifies $x_1 \geq 1$ with $\omega = \{x_1 \mapsto 1\}$
	`red +1 x1 +1 x2 >= 1; x1 x2 x2 x1`	verifies $x_1 + x_2 \geq 1$ with $\omega = \{x_1 \mapsto x_2, x_2 \mapsto x_1\}$
RUP	`rup +1 x1 +1 x2 >= 1;`	verifies $x_1 + x_2 \geq 1$ with RUP
move to core	`core id 3`	moves (3) to the core constraints
deletion from core	`delc 3`	deletes (3) from the core constraints
objective function update	`obju new +1 x1 +1 x2 1;`	defines $x_1 + x_2 + 1$ as new objective
	`obju diff +1 ~x1;`	adds $\overline{x}_1$ to the objective

be used in other presolver reasoning. This scenario arises also in, e.g., MaxSAT solving, and can be dealt with by deriving two PB constraints $f \geq f'$ and $f' \geq f$ in the proof, which encodes that the old and new objective are equal [BBN+23]. Whenever the solver argues in terms of f', a telescoping-sum argument with $f' = f$ can be used to justify the same conclusion in terms of the old objective.

However, if the presolver changes f to f' and then uses reasoning that needs to be certified by redundance-based strengthening, then tricky problems can arise. One of the required proof goals in (1) is that the witness ω cannot worsen the objective. If ω does not mention variables in f', then this is obvious to the presolver—ω has no effect on the objective—but if ω assigns variables in the original objective f, then one still needs to derive $f \geq f\upharpoonright_\omega$ in the formal proof, which can be challenging. While this can often be done by enlarging the witness ω to include earlier variable fixings and identifications, the extra bookkeeping required for this quickly becomes a major headache, and results in the proof deviating further and further from the actual presolver reasoning that the proof logging is meant to certify.

For this reason, a better solution is to introduce a new *objective function update rule* that formally replaces f by a new objective f', so that all future reasoning about the objective can focus on f' and ignore f. Such a rule needs to be designed with care, so that the optimal value of the problem is preserved. Due to space constraints we cannot provide a formal proof here, but recall that intuitively we maintain the invariant for the core set C that it has the same optimal value as the original problem. In agreement with this, the formal requirement for updating the objective from f to f' is to present in the proof log derivations of the two constraints $f \geq f'$ and $f' \geq f$ from the core set C only.

3 Certifying Presolve Reductions

We now describe how feasibility- and optimality-based presolving reductions can be certified by using VeriPB proof logging enhanced with the new objective function update rule described in Sec. 2.2 above. We distinguish between *primal* and *dual* reductions, where primal reductions strengthen the problem formulation by tightening the convex hull of the problem and preserve all feasible solutions, and dual reductions may additionally remove feasible solutions using optimality-based arguments. More precisely, *weak* dual reductions preserve all optimal solutions, but may remove suboptimal solutions. *Strong* dual reductions may remove also optimal solutions as long as at least one optimal solution is preserved in the reduced problem. Our selection of methods is motivated by the recent MIP solver implementation described in [PaP]. Before explaining the individual presolving techniques and their certification, we introduce a few general techniques that are needed for the certification of several presolving methods.

3.1 General Techniques

Substitution. In order to reduce the number of variables, constraints, and non-zero coefficients in the constraints, many presolving techniques first try to identify an equality $E \doteq x_k = \sum_{j \neq k} \alpha_j x_j + \beta$ with $\alpha_j, \beta \in \mathbb{Q}$. Subsequently, all occurrences of x_k in the objective and constraints besides E are substituted by the affine expression on the right-hand side and x_k is removed from the problem. The simplest case when x_k is fixed to zero or one, i.e., when $\beta \in \{0,1\}$ and all $\alpha_j = 0$, is straightforward to handle by deriving a new lower or upper bound on x_k. During presolving, every fixed variable is removed from the model. In the cases where some $\alpha_j \neq 0$, first the equation is expressed as a pair of constraints $E_{\geq} \wedge E_{\leq}$ and then the variable is removed by aggregation as follows.

Aggregation. In order to substitute variables or reduce the number of non-zero coefficients, certain presolving techniques add a scaled equality $s \cdot E \doteq s \cdot E_{\geq} \wedge s \cdot E_{\leq}$, $s \in \mathbb{Q}$, to a given constraint D. We call this an *aggregation*. Since VeriPB certificates expect inequalities with integer coefficients, s is split into two integer scaling factors $s_E, s_D \in \mathbb{Z}$ with $s = s_E/s_D$. In the certificate, the aggregation is expressed as a newly derived constraint

$$D_{new} \doteq \begin{cases} |s_E| \cdot E_{\geq} + |s_D| \cdot D & \text{if } \frac{s_D}{s_E} > 0 \\ |s_E| \cdot E_{\leq} + |s_D| \cdot D & \text{otherwise} \end{cases} .$$

Note that the presolving algorithm may decide to keep working with the constraint $(1/s_D)D_{new}$ internally. In this case, it must store the scaling factor s_D in order to correctly translate between its own state and the state in the certificate; this happens in the implementation used in Sec. 4.

Checked Deletion. The derivation of a new constraint D_{new} can render a previous constraint D redundant. A typical example is the case of substituting a variable above. In a (pre)solver, the previous constraint is overwritten, and in order to keep the constraint database in the proof aligned with the solver, one may want to delete the previous constraint from the proof. In order to check the deletion of D, a subproof is required that proves its redundancy. In most cases, this subproof contains the "inverted" derivation of D_{new}. As an example, consider an aggregation $D_{new} \doteq D + E_{\leq}$ with an equality $E \doteq E_{\leq} \wedge E_{\geq}$. In this case, the subproof for the checked deletion is $D_{new} + E_{\geq}$. Unless stated otherwise, the new constraints are moved to the core and redundant constraints are always removed by inverting the derivation of the constraint that replaces them.

3.2 Primal Reductions

Primal reductions can be certified purely by implicational reasoning.

Bound Strengthening. This preprocessor [FM05, Sav94] tries to tighten the variable domains by iteratively applying well-known *constraint propagation* to all variables in the linear constraints. Each reduced variable domain is communicated to the affected constraints and may trigger further domain changes. This process is continued until no further domain reductions happen or the problem becomes infeasible due to empty domains. Specifically, for an inequality constraint

$$\sum_{j \in N} a_j x_j \geq b \tag{2}$$

with $a_k \neq 0$, we first underestimate $a_k x_k$ via

$$a_k x_k \geq b - \sum_{j \neq k} a_j x_j \geq b - \sum_{j \neq k, a_j > 0} a_j \, .$$

If $a_k > 0$, this yields the lower bound

$$x_k \geq \left\lceil \left(b - \sum_{j \neq k, a_j > 0} a_j\right) / a_k \right\rceil, \tag{3}$$

and if $a_k < 0$ we can obtain an analogous upper bound on x_k.

The bound change can be proven either by RUP, or more explicitly by stating the additions and division needed to form (3) from (2) and the bound constraints. We analyze the effect of both variants in Sec. 4.4.

Parallel Rows. Two constraints C_j and C_k are parallel if a scalar $\lambda \in \mathbb{R}^+$ exists with $\lambda(a_{j1}, \ldots, a_{jn}, b_j) = (a_{k1}, \ldots, a_{kn}, b_k)$. Hence, one of these constraints is redundant and can be removed from the model [ABG+19, GCW+20]. The subproof for deleting the redundant rows must contain the remaining parallel row and λ to prove the redundancy. For a fractional λ the two constraint are scaled to ensure integer coefficients in the certificate.

Probing. The general idea of *probing* [Ach07, Sav94] is to tentatively fix a variable x_j to 0 or 1 and then apply constraint propagation to the resulting model. Suppose

x_k is an arbitrary variable with $k \neq i$, then we can learn fixings or implications in the following cases:

1. If $x_j = 0$ implies $x_k = 1$ and $x_j = 1$ implies $x_k = 0$ we can add the constraint $x_j = 1 - x_k$. Analogously, we can derive $x_k = x_j$ in the case that $x_j = 0$ implies $x_k = 0$ and $x_j = 1$ implies $x_k = 1$.

2. If $x_j = 0$ propagates to infeasibility we can fix $x_j = 1$. Analogously, if $x_j = 1$ propagates to infeasibility we can fix $x_j = 0$.

3. If $x_j = 0$ implies $x_k = 0$ and $x_j = 1$ implies $x_k = 0$ we can fix x_k to 0. Analogously, x_k can be fixed to 1 if $x_j = 0$ implies $x_k = 1$ and $x_j = 1$ implies $x_k = 1$.

Cases 1 and 2 can be proven with RUP. To prove correctness of fixing $x_k = 1$ in Case 3 we first derive two new constraints $x_k + x_j \geq 1$ and $x_k - x_j \geq 0$ in the proof log by RUP. Adding these two constraints leads to $x_k \geq 1$. To prove $x_k = 0$ we derive the constraints $x_k + x_j \leq 0$ and $x_k - x_j \leq 0$ leading to $x_k = 0$.

Simple Probing. On equalities with a special structure, a more simplified version of probing called *simple probing* [ABG+19, Sec. 3.6] can be applied. Suppose the equation

$$\sum_{j \in N} a_j x_j = b \text{ with } \sum_{j \in N} a_j = 2 \cdot b \text{ and } |a_k| = \sum_{j \in N, a_j > 0} a_j - b$$

holds for a variable x_k with $a_k \neq 0$. Let $\hat{N} = \{p \in N \mid a_p \neq 0\}$. Under these conditions, $x_k = 1$ implies $x_p = 0$ and $x_k = 0$ implies $x_p = 1$ for all $p \in \hat{N}$ with $a_p > 0$. Further, $x_k = 1$ implies $x_p = 1$ and $x_k = 0$ implies $x_p = 0$ for all $p \in \hat{N}$ with $a_p < 0$. These implications can be expressed by the constraints

$$x_k = 1 - x_p \text{ for all } p \in \hat{N} \text{ with } a_p > 0\,, \tag{4}$$

$$x_k = x_p \text{ for all } p \in \hat{N} \text{ with } a_p < 0\,. \tag{5}$$

The constraints (4) and (5) can be proven with RUP and used to substitute variables x_p for all $p \in \hat{N}$ from the problem.

Sparsifying the Matrix. The presolving technique sparsify [ABG+19, CM93] tries to reduce the number of non-zero coefficients by adding (multiples of) equalities to other constraints using aggregations. This can be certified as described in Sec. 3.1.

Coefficient Tightening. The goal of this MIP presolving technique, which goes back to [Sav94], is to tighten the LP relaxation, i.e., the relaxation obtained when the integrality requirements are replaced by $x_j \in [0, 1]$. To this end, the coefficients of constraints are modified such that LP relaxation solutions are removed, but all integer feasible solutions are preserved. Suppose we are given a constraint $\sum_{j \in N} a_j x_j \geq b$ with $a_k \geq \varepsilon := a_k - b + \sum_{j \neq k, a_j < 0} a_j > 0$, then the constraint can be strengthened to $(a_k - \varepsilon)x_k + \sum_{j \neq k} a_j x_j \geq b$. The case $a_k < 0$ is handled analogously. This technique is also known as *saturation* in the SAT community [CK05] and VeriPB provides a dedicated saturation rule that can be used directly for proving the correctness of coefficient tightening. The deletion of the original, weaker constraint can be proven automatically.

GCD-based Simplification. This presolving technique from [Wen16] uses a divisibility argument to first eliminate variables from a constraint and then tighten its right-hand side. Given $C \doteq \sum_{j \in N} a_j x_j \geq b$ with $|a_1| \geq \cdots \geq |a_n| > 0$. We define the greatest common divisor $g_k = \gcd(a_1, \ldots, a_k)$ as the largest value g such that $a_j/g \in \mathbb{Z}$ for all $j \in \{1, \ldots, k\}$. If for an index k it holds that $b - g_k \cdot \left\lceil \frac{b}{g_k} \right\rceil \geq \sum_{k<j\leq n, a_j>0} a_j$ and $b - g_k \cdot \left\lceil \frac{b}{g_k} \right\rceil - g_k \leq \sum_{k<j\leq n, a_j<0} a_j$, then all $a_{k+1}, \ldots, a_n$ can be set to 0. This first step can be certified as *weakening* [LBMW20] and VeriPB provides an out-of-the-box verification function for it. Finally, b can be rounded to $g_k \cdot \lceil b/g_k \rceil$. This rounding step can be certified by dividing C with g_k and then multiply it again with g_k.

Substituting Implied Free Variables. A variable x_j is called *implied free* if its lower bound and its upper bound can be derived from the constraints. For example, the constraints $x_1 - x_2 \geq 0$ and $x_2 \geq 0$ imply the lower bound $x_1 \geq 0$. If we have an implied free variable x_j in an equality $E \doteq a_j x_j + \sum_{k \neq j} a_k x_k = b$ with $a_j > 0$, then we can remove x_j from the problem by substituting it with $x_j = \left(b - \sum_{k \neq j} a_k x_k\right)/a_j$, see [ABG+19] for details.

To apply the substitution in the certificate we use aggregations to remove x_j from all constraints and the objective function update to remove x_j from the objective. If coefficients c_j/a_j or a_k/a_j are non-integer, then the resulting constraints are scaled as described in Sec. 3.1. To prove the deletion of E, we derive two constraints by adding $x_j \geq 0$ and $1 \geq x_j$ to E each, which results in

$$b \geq \sum_{k \neq j} a_k x_k \ \wedge \ \sum_{k \neq j} a_k x_k \geq b - a_j \,. \tag{6}$$

Then the deletion of $E_{\geq}$ can be certified by a witness $\omega = \{x_j \mapsto 1\}$. The constraint simplifies to (6) and is therefore fulfilled. Analogously, we use the witness $\omega = \{x_j \mapsto 0\}$ to certify the deletion of $E_{\leq}$. Finally, to delete the constraints in (6) we generate a subproof that shows that negation of the auxiliary constraints in (6) leads to $x_j \notin \{0, 1\}$. This is a contradiction to the implied variable bounds $0 \leq x_j \leq 1$. Since these bounds are still present through the implying constraints, we can add these implying constraints to (6) in the subproof to arrive at a contradiction.

Singleton Variables. It is well-known that variables that appear only in one inequality constraint or equality can be removed from the problem [ABG+19, Sec. 5.2]. This can be certified by applying one of the following primal or dual strategies in this order: First, try to apply duality-based fixing, see Sec. 3.3; second, an implied free singleton variable can be substituted as explained above; otherwise, the singleton variable can be treated as a *slack variable*: substitute the variable in the objective, then relax the equality as in (6), and delete the original constraint.

3.3 Dual Reductions

Dual reductions remove solutions while preserving at least one optimal solution. Hence, to prove the correctness of dual reductions we need to involve the redundance-based strengthening rule of VeriPB. For each derived constraint C we only explain how to prove $f \geq f\restriction_\omega$ (subject to the negation $\neg C$); the proof goals for $C\restriction_\omega$ can be derived in a very similar fashion.

Duality-based Fixing. This presolving step described in [ABG+19, Sec. 4.2] counts the *down-* and *up-lock* of a variable. A down-lock on variable x_j is a negative coefficient, an up-lock on variable x_j is a positive coefficient (for $\geq$ constraints). If x_j has no down-locks and $c_j \leq 0$, it can be fixed to zero; if x_j has no up-locks and $c_j \geq 0$, it can be fixed to one. These reductions can be certified with redundance-based strengthening using the witness $\omega = \{x_j \mapsto v\}$, where v is the fixing value. The proof goal for $f \geq f\restriction_\omega$ is equivalent to $c_j x_j \geq c_j v$, which is fulfilled by the conditions of duality-based fixing.

Dominated Variables. A variable x_j is said to *dominate* another variable x_k [ABG+19, GKM+15], in notation $x_j \succ x_k$, if

$$c_j \leq c_k \ \wedge \ a_{ij} \geq a_{ik} \text{ for all } i \in \{1, \dots, m\}\,, \tag{7}$$

where a_{ij} and a_{ik} are the coefficients of variable x_j and x_k, respectively, in the i-th constraint. Variable x_j is then favored over x_k since x_j contributes less to the objective function, but more to the feasibility of the constraints. For every domination $x_j \succ x_k$, a constraint $C \doteq x_j \geq x_k$ can be introduced. This constraint can be certified by redundance-based strengthening with the witness $\omega = \{x_k \mapsto x_j, x_j \mapsto x_k\}$. The proof goal for $f \geq f\restriction_\omega$ is equivalent to

$$c_j x_j + c_k x_k \geq c_j x_k + c_k x_j\,. \tag{8}$$

The negated constraint $\neg C \doteq x_j < x_k$ leads to $x_k = 1$ and $x_j = 0$. Substituting these values in (8) leads to $c_k \geq c_j$, which follows directly from Condition (7).

Dominated Variables Advanced. For an implied free variable we can drop the variable bounds and pretend the variable is unbounded. This allows for additional fixings in the following cases of dominated variables:

(a) If the upper bound of x_j is implied and $x_j \succ x_k$, then $x_k = 0$.

(b) If the lower bound of x_k is implied and $x_j \succ x_k$, then $x_j = 1$.

(c) If the upper bound of x_j is implied and $x_j \succ -x_k$, then $x_k = 1$.

(d) If the lower bound of x_j is implied and $-x_j \succ x_k$, then $x_j = 0$.

We use redundance-based strengthening with witness $\omega = \{x_k \mapsto 0\}$ to prove the correctness of a as follows. If the upper bound of x_j is implied, this means there exists a constraint with $a_{ij} < 0$ such that $x_j \leq \left\lfloor \frac{b_\ell - \sum_{\ell \neq j, a_{i\ell} > 0} a_{i\ell}}{a_{ij}} \right\rfloor = 1$. Due to Condition (7), it must hold that $0 > a_{ij} \geq a_{ik}$, and the constraint $x_j + x_k \leq 1$ can be derived. Hence, negating and propagating $C \doteq x_k = 0$ with RUP leads to contradiction, which proves the validity of C. Case b can be handled analogously using the witness $\omega = \{x_k \mapsto 1\}$. To derive $C \doteq x_k = 1$ in c we use redundance-based strengthening with witness $\omega = \{x_k \mapsto 1, x_j \mapsto 1\}$. Then, the proof goal for $f \geq f\restriction_\omega$ is $c_j \cdot x_j + c_k \cdot x_k \geq c_j + c_k$. After propagating $\neg C$, this becomes equivalent to $c_j \leq -c_k$, which is true by Condition (7). Case d can be handled analogously using the witness $\omega = \{x_k \mapsto 0, x_j \mapsto 0\}$.

3.4 Example

We conclude this section with an example of a small certificate for the substitution of an implied free variable in Fig. 1, also available with a more detailed description at the software repository of PaPILO [Hoe23]. Consider the 0–1 ILP

$$\min\ x_1 + x_2 \ \text{ s.t. }\ x_1 + x_2 - x_3 - x_4 = 1\,, \tag{9}$$

$$-x_1 + x_5 \geq 0\,, \tag{10}$$

in which the lower bound of x_1 is implied by (9) and the upper bound of x_1 is implied by (10). Hence, x_1 is implied free and we can use (9) to substitute it.

In the left section of Fig. 1 we first derive the two auxiliary constraints

$$0 \leq x_2 - x_3 - x_4 \leq 1\,, \tag{11}$$

which receives the constraint IDs 4 and 5 and are moved to the core. Note that the equality in (9) is split into two inequalities with IDs 1 and 2. In the middle section, we first remove x_1 from (10) by aggregation with (9), perform checked deletion, then remove x_1 from the objective (automatically proven by VeriPB). Last, in the right section, we delete the equality in (9) used for the substitution and the auxiliary constraints in (11) and arrive at the reformulated problem min $x_3 + x_4 + 1$ s.t. $x_2 - x_3 - x_4 + x_5 \geq 1$. From here, we could continue to derive $x_2 = 1$ by duality-based fixing, since x_2 has zero up-locks and objective coefficient zero. This displays the importance of the objective update, as without it x_2 would still contribute to the objective with a positive coefficient, and this would prohibit duality-based fixing to 1.

4 Computational Study

In this section we quantify the cost of *certifying* presolve reductions in a state-of-the-art implementation for MIP-based presolve (Sec. 4.2) and the cost of *verifying* the resulting certificates (Sec. 4.3). In Sec. 4.4, we analyze the impact of certifying constraint propagation by RUP or by an explicit cutting planes proof.

```
* generates ID 4:
pol 1 ~x1 + ;
core id 4
* generates ID 5:
pol 2 x1 + ;
core id 5
```

```
* generates ID 6:
pol 3 1 + ;
core id 6
delc 3 ;  ; begin
   pol 6 2 +
end
obju new +1 x3 +1 x4 1 ;
```

```
delc 2 ; x1 -> 0
delc 1 ; x1 -> 1
delc 5
delc 4 ; ; begin
   pol 6 -1 +
end
```

Figure 1: *A VeriPB certificate to substitute an implied free variable x_1.*

4.1 Experimental Setup

For generating the presolve certificates we use the solver-independent presolve library PaPILO [PaP], which provides a large set of MIP and LP techniques from the literature, described in Sec. 3. Additionally, it accelerates the search for presolving reductions by parallelization, encapsulating each reduction in a so-called transaction to avoid expensive synchronization [GGH23]. Logging the certificate, however, is performed sequentially while evaluating the transactions.

We base our experiments on models from the Pseudo-Boolean Competition 2016 [Rou16] including 1398 linear small integer decision and 532 linear small integer optimization instances of the competitions PB10, PB11, PB12, PB15, and PB16 and 295 decision and 145 optimization instances from MIPLIB 2017 [GHG$^+$21] in the OPB translation [Dev20], excluding 10 large-scale instances[1] for which PaPILO reaches the memory limit. This yields a total of 671 optimization and 1681 decision instances. We use PaPILO 2.2.0 [HG23] running on 6 threads and VeriPB 2.0 [GO23]. The experiments are carried out on identical machines with an 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60 GHz CPU and 16 GB of memory and are assigned 14,000 MB of memory. The strict time limit for presolve plus certification and verification is three hours. Times (reported in seconds) do not include the time for reading the instance file. For all aggregations, we use the shifted geometric mean with a shift of 1 second.

4.2 Overhead of Proof Logging

In the first experiment, we analyze the overhead of proof logging in PaPILO. The average results are summarized in Tab. 2, separately over decision (dec) and optimization (opt) instances for PB16 and MIPLIB. Column "relative" indicates the average slow-down incurred by printing the certificate.

The relative overhead of proof logging is less than 6% across all test sets. VeriPB supports two variants to change the objective function. Either printing the entire objective (`obju new`) or printing only the changes in the objective (`obju diff`). In our experiments, we only print the changes, since printing the entire objective for each change can lead to a large certificate and overhead, especially for instances with large and dense objective functions. On the PB16 instance normalized-

[1]normalized-184, normalized-pb-simp-nonunif, a2864-99blp, ivu06-big, ivu59, supportcase11, a2864-99blp.0.s/u, supportcase11.0.s/u

Table 2: *Runtime comparison of PaPILO with and without proof logging.*

test set	size	default [s]	w/proof log [s]	relative
PB16-dec	1397	0.06	0.06	1.00
MIPLIB-dec	291	0.42	0.43	1.02
PB16-opt	531	0.65	0.66	1.02
MIPLIB-opt	142	0.33	0.35	1.06

Table 3: *Time to verify the certificates. VeriPB timeouts are treated with PAR2.*

			PaPILO time [s]		VeriPB	relative time w.r.t.	
test set	size	verified	default	w/proof log	time [s]	default	w/proof log
PB16-dec	1397	1397	0.06	0.06	0.88	14.67	14.67
MIPLIB-dec	291	267	0.42	0.43	9.64	22.85	22.42
PB16-opt	531	520	0.65	0.66	10.44	16.06	15.82
MIPLIB-opt	142	139	0.33	0.35	5.25	15.91	15.00

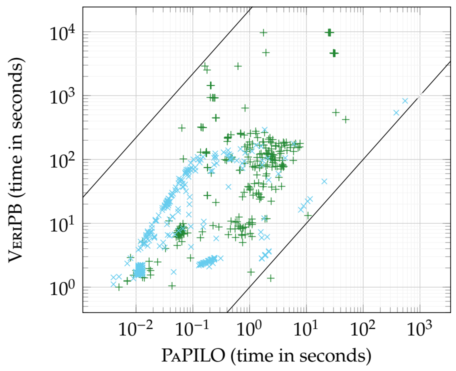
datt256, for example, PaPILO finds 135 206 variable fixings. Updating the entire objective function with 262 144 non-zeros for each of these variables leads to a huge certificate of about 138 GB and increases the time from 3.3 seconds (when printing only the changes) to 6625 seconds.[2]

For 99% of the instances, we can further observe that the *overhead per applied reduction* is below $0.001 \cdot 10^{-3}$ seconds over both test sets. This means that the proof logging overhead is not only small on average, but also small per applied reduction on the vast majority of instances. These results show that the overhead scales well with the number of applied reductions and that proof logging remains viable even for instances with many transactions. Here, under applied reductions we subsume all applied transactions and each variable fixing or row deletion in the first model clean-up phase. During model clean-up, PaPILO fixes variables and removes redundant constraints from the problem. While PaPILO technically does not count these reductions as full transactions found during the parallel presolve phase, their certification can incur the same overhead.

4.3 Verification Performance on Presolve Certificates

In this section, we analyze the time to verify the certificates generated by PaPILO. The results are summarized in Tab. 3. The "verified" column lists the number of instances verified within 3 hours. VeriPB timeouts are counted as twice the time limit, i.e., PAR2 score. Similar to Tab. 2, the "relative" columns report the relative overhead of VeriPB runtime compared to PaPILO.

[2]Certificate generated on Intel Xeon Gold 5122 @ 3.60GHz 96 GB with 50,000 MB of memory assigned.



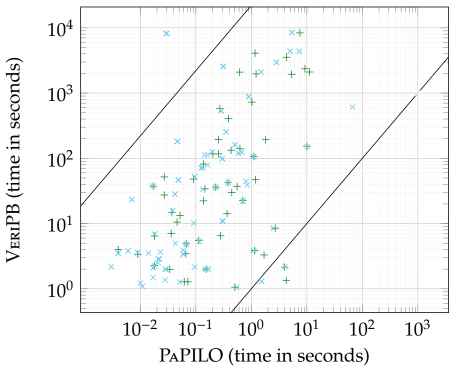


Figure 2: *Running times of VeriPB vs. PaPILO on test sets PB16 (left) and MIPLIB (right), including all instances with more than 1 seconds in VeriPB and less than 30 minutes in PaPILO, and excluding timeouts. Green + signs mark optimization and blue × signs mark decision instances.*

First note that all certificates are verified by VeriPB (partially on the 38 instances where VeriPB times out). On average, it takes between 14.7 and 22.4 times as much time to verify the certificates than to produce them. Nevertheless, some instances take a longer than average time to verify. Over all test sets, 25% of the instances have an overhead of at least a factor of 193, see also Fig. 2.

To put this result into context, note that presolving amounts more to a transformation than to a (partial) solution of the problem. Each reduction has to be certified and verified while a purely solution-targeted algorithm may be able to skip certifying of a larger part of the findings that are not form a part of the final proof of optimality. Hence, it makes sense to compare the performance of VeriPB on presolve certificates to the overhead for, e.g., for verifying CNF translations [GMNO22]. For this study, a similar performance overhead is reported as in Fig. 2.

4.4 Performance Analysis on Constraint Propagation

Finally, we investigate how the performance of VeriPB depends on whether we use RUP (as in Sec. 4.2 and Sec. 4.3) or explicit cutting planes derivations (POL) to certify bound strengthening reductions from constraint propagation. Here, we additionally exclude 9 large-scale instances[3] for which PaPILO reaches the memory limit when certifying with POL. The results are summarized in Tab. 4. The "verified" column contains the number of instances verified by VeriPB within the time limit. The "time" column reports the time for verification.

Deriving the propagation directly with cutting planes is 3.2% faster on PB16-dec, 2.8% faster on MIPLIB-dec, 13.1% faster on MIPLIB-opt, and 0.7% faster on

[3] neos-4754521-awarau.0.s, neos-827015.0.s/u, neos-829552.0.s/u, s100.0.s/u, normalized-datt256, s100

Table 4: *Comparison of the runtime of VeriPB with RUP and POL over instances with at least 10 propagations.*

		RUP		POL		
test set	size	verified	time [s]	verified	time [s]	relative
PB16-dec	284	284	2.21	284	2.14	0.968
MIPLIB-dec	35	31	153.23	31	148.88	0.972
PB16-opt	153	142	28.43	142	28.22	0.993
MIPLIB-opt	16	14	147.11	14	127.83	0.869

PB16-opt. On 95% of the decision instances using RUP is at most 9.7% slower. While it is expected that verification is faster when the cutting planes proof is given explicitly, it is surprising that the performance difference between the methods is not more pronounced.This is partly due to the cost of the watched-literal scheme [MMZ+01, SS06] used by VeriPB for unit propagation. The overhead of maintaining the watches is present regardless of whether (reverse) unit propagation is used or not. Furthermore, unit propagation is also used for automatically verifying redundance-based strengthening. Together, this limits the potential for runtime savings by providing the explicit cutting planes proof.

Furthermore, providing an explicit cutting planes proof for propagation requires printing the constraint into the certificate. Hence, the certificate size becomes dependent on the number of non-zeros in the constraints leading to propagations. In contrast, the overhead of RUP is constant and much smaller.

All in all, these results suggest to prefer RUP when deriving constraint propagation since it barely impacts the performance of VeriPB and keeps the size of the certificate smaller. The computational cost of RUP could be further reduced by extending it to accept an ordered list of constraints that shall be propagated first, similar as in [CFHH+17]. Such an extension could also be used for other presolving techniques, in particular probing and simple probing.

5 Conclusion

In this paper we set out to demonstrate how presolve techniques from state-of-the-art MIP solvers can be equipped with certificates in order to verify the equivalence between original and reduced models. Although the pseudo-Boolean proof logging format behind VeriPB [BGMN22] was not designed with this purpose in mind, we could show that a limited extension needed for handling updates of the objective function is sufficient to craft a certified presolver for 0–1 ILPs.

However, our experimental study on instances from pseudo-Boolean competitions and MIPLIB also exhibited that the verification of MIP-based presolving can suffer from large and overly verbose certificates. To shrink the proof size we introduced a sparse objective update function but identified further possible improvements. First, a native substitution rule in VeriPB would remove the need for the explicit derivation of new aggregations and the verification of checked deletion as described in Sec. 3.1. For instances where presolving is dominated by substitutions, we estimate that this would reduce certificate sizes by up to 90%, and no more time would be spent on checked deletion for substitutions. Second, augmenting the RUP syntax by the option to specify an ordered list of constraints to propagate first, similarly as in [CFHH+17], would accelerate RUP, in particular for fast verification of bound strengthenings by constraint propagation.

While VeriPB is currently restricted to operate on integer coefficients only, the certification techniques presented in Sec. 3 do not rely on this assumption and are applicable to general binary programs. It has been shown how to construct VeriPB certificates for bounded integer domains [GMN22, MM23], and within the framework of the generalized proof system laid out in [DEGH23], our certificates would even translate to continuous and unbounded integer domains. To conclude, we believe our results show convincingly that this type of proof logging techniques is a very promising direction of research also for MIP presolve beyond 0–1 ILPs.

Acknowledgements

The authors wish to acknowledge helpful technical discussions on VeriPB in general and the objective update rule in particular with Bart Bogaerts, Ciaran McCreesh, and Yong Kiam Tan. The work for this article has been partly conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF grant number 05M14ZAM). 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. The computational experiments were enabled by resources provided by LUNARC at Lund University.

References

[ABG+19] Tobias Achterberg, Robert Bixby, Zonghao Gu, Edward Rothberg, and Dieter Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 32, 11 2019.

[Ach07] Tobias Achterberg. *Constraint Integer Programming*. Doctoral thesis, Technische Universität Berlin, Fakultät II - Mathematik und Naturwissenschaften, Berlin, 2007.

[AGJ+18] 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.

[BBN$^+$23] 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.

[BGMN22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI '22)*, pages 3698–3707, February 2022.

[BGMN23] 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*.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Biere et al. [BHvMW21], chapter 9, pages 391–435.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFHH$^+$17] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[CGS17] 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.

[CHH$^+$17] 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.

[CK05] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, March 2005. Preliminary version in *DAC '03*.

[CKSW13a] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CKSW13b] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, 2013.

[CM93] S. Frank Chang and S. Thomas McCormick. Implementation and computational results for the hierarchical algorithm for making sparse matrices sparser. *ACM Trans. Math. Softw.*, 19(3):419–441, sep 1993.

[CMS17] 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.

[DEGH23] Jasper van Doornmalen, Leon Eifler, Ambros Gleixner, and Christopher Hojny. A proof system for certifying symmetry and optimality reasoning in integer programming. Technical Report 2311.03877, arXiv.org, November 2023.

[Dev20] Jo Devriendt. Miplib 0-1 instances in opb format. 05 2020.

[EG22] Leon Eifler and Ambros Gleixner. A computational status update for exact rational mixed integer programming. *Mathematical Programming*, 2022.

[EG24] Leon Eifler and Ambros Gleixner. Safe and verified gomory mixed integer cuts in a rational MIP framework. *SIAM Journal on Optimization*, 34(1):742–763, 2024.

[EGP22] Leon Eifler, Ambros Gleixner, and Jonad Pulaj. A safe computational framework for integer programming applied to chvátal's conjecture. *ACM Transactions on Mathematical Software*, 48(2), 2022.

[FM05] Armin Fügenschuh and Alexander Martin. Computational integer programming and cutting planes. In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks* *in Operations Research and Management Science*, pages 69–121. Elsevier, 2005.

[GCW$^+$20] Patrick Gemander, Wei-Kun Chen, Dieter Weninger, Leona Gottwald, and Ambros Gleixner. Two-row and two-column mixed-integer presolve using hashing-based pairing methods. *EURO Journal on Computational Optimization*, 8(3-4):205 – 240, 2020.

[GGH23] Ambros Gleixner, Leona Gottwald, and Alexander Hoen. PaPILO: A parallel presolving library for integer and linear programming with multiprecision support. *INFORMS Journal on Computing*, 2023.

[GHG$^+$21] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 13:443–490, 2021.

[GKM$^+$15] Gerald Gamrath, Thorsten Koch, Alexander Martin, Matthias Miltenberger, and Dieter Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7, 06 2015.

[GMM$^+$20] 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.

[GMN20] 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.

[GMN22] 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.

[GMNO22] 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.

[GN21] 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.

[GO23] Stefan Gocht and Andy Oertel. Veripb, 2023. githash: dd7aa5a1.

[GSD19] 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.

[HG23] Alexander Hoen and Leona Gottwald. Papilo: Parallel presolve integer and linear optimization, 2023. githash: 3b082d4.

[HHW13a] 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.

[HHW13b] 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.

[Hoe23] Alexander Hoen. Papilo: Parallel presolve integer and linear optimization, 2023. githash: 5df3dd6d.

[Klo14] Ed Klotz. Identification, assessment, and correction of ill-conditioning and numerical instability in linear and integer programs. In Alexandra Newman and Janny Leung, editors, *Bridging Data and Decisions*, TutORials in Operations Research, pages 54–108. 2014.

[LBMW20] Daniel Le Berre, Pierre Marquis, and Romain Wallon. On weakening strategies for pb solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 322–331, Cham, 2020. Springer International Publishing.

[MM23] 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.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver.

In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[PaP] PaPILO — parallel presolve for integer and linear optimization. `https://github.com/lgottwald/PaPILO`.

[Rou16] Olivier Roussel. Pseudo-boolean competition 2016, 2016.

[Sav94] Martin Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6, 11 1994.

[SBD19] Youcef Sahraoui, Pascale Bendotti, and Claudia D'Ambrosio. Real-world hydro-power unit-commitment: Dealing with numerical errors and feasibility issues. *Energy*, 184:91–104, 2019. Shaping research in gas-, heat- and electric- energy infrastructures.

[SS06] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):165–189, March 2006. Preliminary version in *DATE '05*.

[Ste11] Daniel E. Steffy. *Topics in exact precision mathematical programming*. PhD thesis, Georgia Institute of Technology, 2011.

[VDB22] 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 (LP-NMR '22)*, volume 13416 of *Lecture Notes in Computer Science*, pages 429–442. Springer, September 2022.

[Wen16] Dieter Weninger. *Solving mixed-integer programs arising in production planning*. Phd thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016.

[WHH14] 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.

End-to-End Verification for Subgraph Solving

Abstract

Modern subgraph-finding algorithm implementations consist of thousands of lines of highly optimized code, and this complexity raises questions about their trustworthiness. Recently, some state-of-the-art subgraph solvers have been enhanced to output machine-verifiable proofs that their results are correct. While this significantly improves reliability, it is not a fully satisfactory solution, since end-users have to trust both the proof checking algorithms and the translation of the high-level graph problem into a low-level 0–1 integer linear program (ILP) used for the proofs.

In this work, we present the first *formally verified* toolchain capable of full end-to-end verification for subgraph solving, which closes both of these trust gaps. We have built encoder frontends for various graph problems together with a 0–1 ILP (a.k.a. pseudo-Boolean) proof checker, all implemented and formally verified in the CakeML ecosystem. This toolchain is flexible and extensible, and we use it to build verified proof checkers for both decision and optimization graph problems, namely, *subgraph isomorphism*, *maximum clique*, and *maximum common (connected) induced subgraph*. Our experimental evaluation shows that end-to-end formal verification is now feasible for a wide range of hard graph problems.

1 Introduction

Combinatorial optimization algorithms have improved immensely since the turn of the millennium, and are now routinely used to solve large-scale real-world problems, through both general-purpose solving paradigms [BHvMW21, BR07, GSVW14] and dedicated algorithms for more specialised problems such as subgraph finding [MPT20]. Since these combinatorial solvers are used for an increas-

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, Febuary 2024.

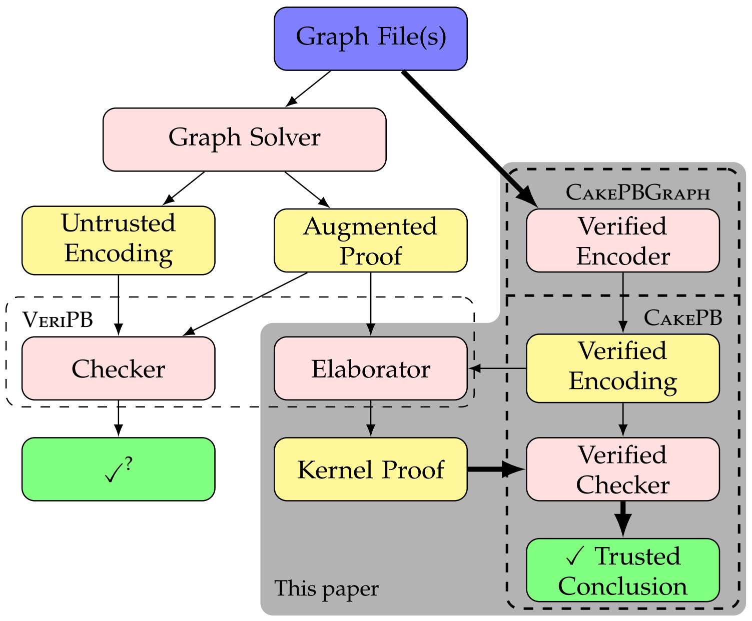


Figure 1: *The full verification workflow. Without verified proof checking, only the left-hand part of the diagram is used. Our current work enables the additional shaded parts, where the thick dashed box is the formally verified program and thick arrows show its key input-output interfaces.*

ingly wide range of applications, it becomes crucial that the results they compute can be trusted. Sadly, this is currently not the case [CKSW13, AGJ$^+$18, GSD19, BMN22]. Extensive testing, though beneficial, has not been able to resolve the problem of solvers occasionally producing faulty answers, and attempts to build correct-by-construction software using formal verification run into the obstacle that current techniques cannot scale to the level of complexity of modern solvers.

Instead, the most promising way to achieve verifiably correct combinatorial solving seems to be *proof logging*, meaning that solvers produce efficiently verifiable certificates of correctness that can be corroborated by an independent proof checking program [MMNS11]. This approach has been successfully used in the SAT community [HHW13a, HHW13b, WHH14], which raises the question of whether similar techniques could be employed in other settings such as subgraph finding. For this it would seem that the proof checker would need to understand graph concepts such as vertices, edges, neighbourhoods, et cetera. Surprisingly, this turns out not to be the case—instead, the solver can encode the graph problem using 0–1 linear inequalities (also referred to as *pseudo-Boolean constraints*), and then justify its complex high-level reasoning in terms of this low-level representation. This approach has been used to add proof logging with the VeriPB tool to state-of-the-art solvers for subgraph isomorphism, clique, and maximum common (connected) induced subgraph [GMN20, GMM^{+}20], as illustrated in the left-hand part of Figure 1. We emphasize that although this approach uses reasoning with pseudo-Boolean constraints for the proof logging, it is *not* limited to pseudo-Boolean solving. Rather, it can be used to certify the output of *any* untrusted solver—such as tools that operate natively on graph representations—as long as the solver's relevant reasoning steps can be expressed with pseudo-Boolean proofs.

While this approach has been successful for debugging solvers and providing convincing demonstrations that the fixed solvers are producing correct answers, it is important to observe that it crucially hinges on the assumption that three components are correct: (1) the low-level encoding of the problem, (2) the proof checker, and (3) the interpretation of the final output. For example, if the maximum clique solver in [GMM^{+}20] produces a proof accepted by the VeriPB checker, then one can conclude that *if* the 0–1 ILP encoding of clique is implemented correctly, and *if* VeriPB does not contain bugs, and *if* (say) a 200-vertex graph having a maximum clique size of 13 corresponds to the optimal objective value for the low-level encoding being 187 (because it minimises the number of vertices not in the clique), then the maximum clique size is indeed 13. Such assumptions are not unreasonable—encodings have been chosen to be as simple as possible and the code can be subjected to extensive testing; the proof format is designed so that proof checking should be easy; and verifying that proof outputs correspond to solver outputs is not too cumbersome. Compared to having to trust an extremely complex solver, this is a vast improvement. However, if provably correct results are the end goal, then this still leaves much to be desired.

1.1 Our Contribution

In this work, we resolve all the concerns discussed above by presenting the first toolchain capable of end-to-end formal verification for state-of-the-art algorithms for maximum clique, subgraph isomorphism, and maximum common (connected) induced subgraph problems. Although the implementations of modern solvers for these problems are far too complicated to be formally verified by current techniques, we can still use formal verification to certify the correctness of the proof logging and proof checking process. We do so by defining a solver-friendly *augmented* VeriPB proof format; enhancing the VeriPB tool with a *proof elaborator* that can translate such augmented proofs to a more explicit *kernel* format; and designing a *formally verified proof checker* for the kernel format. This formally verified checker is also capable of providing its own formally verified encodings from graph problems to 0–1 ILPs. Finally, the output provided by the formally verified proof checker is in terms of the original problem, not the low-level encoding. This means that using the process illustrated in the right-hand part of Figure 1, if the checking process outputs (say)

```
s VERIFIED MAX CLIQUE SIZE |CLIQUE| = 13
```

then we can be absolutely sure that the maximum clique size for our graph is 13, *if* we trust the formal verification tool(s) and *if* the formal higher-order logic

$$\begin{aligned}
&\mathsf{is\_clique}\ vs\ (v,e) \overset{\text{def}}{=} \\
&\quad vs \subseteq \{\ 0,1,\ldots,v{-}1\ \} \land \\
&\quad \forall x\ y.\ x \in vs \land y \in vs \land x \neq y \Rightarrow \mathsf{is\_edge}\ e\ x\ y \\
&\mathsf{max\_clique\_size}\ g \overset{\text{def}}{=} \mathsf{max}_{\mathsf{set}}\ \{\ \mathsf{card}\ vs \mid \mathsf{is\_clique}\ vs\ g\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{has\_subgraph\_iso}\ (v_p,e_p)\ (v_t,e_t) \overset{\text{def}}{=} \\
&\quad \exists f.\ \mathsf{inj}\, f\ \{\ 0,1,\ldots,v_p{-}1\ \}\ \{\ 0,1,\ldots,v_t{-}1\ \} \land \\
&\qquad \forall a\ b.\ \mathsf{is\_edge}\ e_p\ a\ b \Rightarrow \mathsf{is\_edge}\ e_t\ (f\ a)\ (f\ b)
\end{aligned}$$

Figure 2: *HOL definitions for maximum clique size of a graph with v vertices and edge set e (top), and existence of a subgraph isomorphism from a pattern graph* (v_p, e_p) *to a target graph* (v_t, e_t) *(bottom).*

(HOL) specifications (as shown in Figure 2) accurately reflect what it means to be a clique. The toolchain we provide is also flexible and extensible, in that it can be readily adapted to other combinatorial problems, including problems not involving graphs.

1.2 Comparison to Related Work

Formally verified proof checkers have previously played an important role in SAT solving [CMS17, CHH+17, Lam20] and are vital for widespread acceptance of SAT-solver-generated mathematical proofs [HK17]. However, such proof checkers have worked only for conjunctive normal form (CNF), and only to establish that decision problems encoded in CNF are infeasible: verification that the encoding accurately reflects the problem to be solved has either been ignored or has been handled separately, e.g., [CMS19, SFL+21, CAH23]. For graph problems, previous attempts at verified proof checking have been tied to one specific problem, or even one specific algorithm, e.g., [BDM23]. In contrast, we provide formal verification for optimization problems and with much more expressive formats than CNF, and we do so in a unified way with a single pseudo-Boolean proof logging format for 0–1 linear inequalities together with a general-purpose toolchain, rather than having to design proof logging from scratch for each new combinatorial problem considered. In this way, we demonstrate that end-to-end formally verified combinatorial solving is now eminently within reach, by combining pseudo-Boolean proof logging with formally verified tools for 0–1 ILP encodings and pseudo-Boolean proof checking.

1.3 Outline of This Paper

After reviewing preliminaries in Section 2, we describe the formally verified proof checker in Section 3 and how solver proofs in a user-friendly proof format can be converted to a more restricted format accepted by this proof checker in Section 4.

We report results from an experimental evaluation in Section 5. We conclude in Section 6 with a discussion of future research directions.

2 Preliminaries

Our discussion of pseudo-Boolean proof logging will be brief, since the main thrust of this work is how to formally verify proof logging rather than to design it. See [GN21] and [BGMN23] for more on the VeriPB system and [BN21] for background on the cutting planes reasoning method used.

A *literal* ℓ over a variable x is x itself or its negation $\overline{x}$, taking values 0 (false) or 1 (true), so that $\overline{x} = 1 - x$. A *pseudo-Boolean (PB) constraint* C is a 0-1 integer linear inequality $\sum_i a_i \ell_i \geq A$, which without loss of generality we can always assume to be in *normalized form*; i.e., all literals ℓ_i are over distinct variables and the coefficients a_i and the *degree (of falsity)* A are non-negative. The *negation* $\neg C$ of C is $\sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1$ (saying that the sum of the coefficients of falsified literals is so large that the satisfied literals can contribute at most $A - 1$). A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints.

Cutting planes [CCT87] is a method for iteratively deriving new constraints logically implied by a PB formula by taking positive linear combinations or dividing a constraint and rounding up. We say that C *unit propagates* the literal ℓ if under the current partial assignment C cannot be satisfied unless ℓ is set to true, and that C is implied by F by *reverse unit propagation (RUP)* if adding $\neg C$ to F and then unit propagating until saturation leads to contradiction in the form of a violated constraint. VeriPB allows adding constraints by RUP, which is a convenient way of avoiding having to write out explicit syntactic derivations.

In addition to deriving constraints C that are implied by F, VeriPB also has *strengthening* rules for inferring *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. If there is a partial mapping ω of variables to literals and/or truth values such that

$$F \cup \{\neg D\} \vdash (F \cup D)\restriction_\omega \tag{1}$$

holds, meaning that after applying ω to $F \cup \{D\}$ all of the resulting constraints can be derived by cutting planes from $F \cup \{\neg D\}$, then D can be added by *redundance-based strengthening*. There is also a similar but slightly different *dominance-based strengthening* rule. Importantly, the proof has to specify ω and also contain explicit subderivations for all *proof goals* in $(F \cup D)\restriction_\omega$ in Equation (1) unless they are obvious enough that VeriPB can automatically figure them out (e.g., by using RUP). Finally, for optimization problems there are rules to deal with objective functions and incumbent solutions, and the strengthening rules also need to be slightly adapted for this setting.

The formalization of our proof checking toolchain is carried out in the HOL4 proof assistant for classical higher-order logic [SN08]. We make particular use of the CakeML tools for production and optimization of verified CakeML source

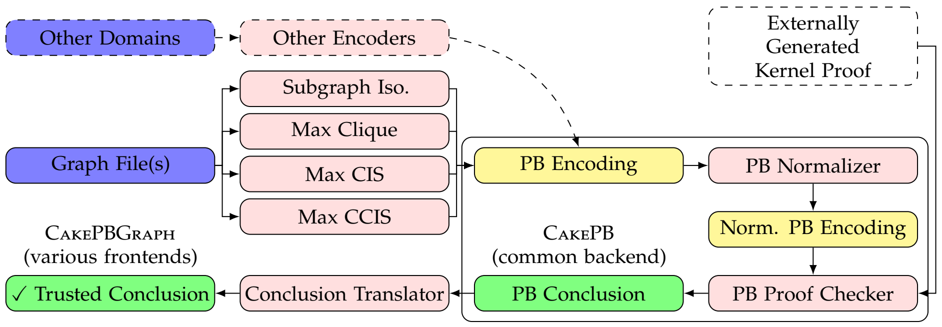


Figure 3: *Architecture of the end-to-end verified proof checkers for various graph problems.*

code [MO14, GMKN17] as well as for formally verified compilation [TMK+19], allowing to transfer guarantees of source-code-level correctness down to executable machine code. Where applicable, formal code snippets are pretty-printed for illustration, e.g., as shown in Figure 2. The set and first-logic notation is standard (e.g., $\Rightarrow$ denotes logical implication); other HOL notation is explained where appropriate. Formally verified results are preceded by a turnstile $\vdash$. All code is available in the supplementary material [GMM+23].

3 Formally Verified Graph Proof Checkers

This section details the formal verification of our pseudo-Boolean proof checker CakePB and its various graph frontends, focusing on the key architectural decisions and reusable insights behind the verification effort. An overview of the tool is shown in Figure 3. We first present the different components, and then plug them together to obtain end-to-end verified graph proof checkers.

3.1 Verified Pseudo-Boolean Proof Checking

A key design objective for CakePB is to make it a general yet effective pseudo-Boolean proof checking backend. To this end, CakePB supports a *kernel* subset of the VeriPB proof format with cutting planes, strengthening, and optimization rules as discussed in Section 2. The implementation and verification of all of this within a single proof checker backend presents several new challenges compared to prior tools for efficient verified CNF proof checking [CHH+17, Lam20, THM23]. Firstly, the pseudo-Boolean proof system features a much richer set of rules, each of which needs a formal soundness justification. Secondly, there is an intricate interplay between different proof rules, especially concerning how they preserve optimal solutions (or satisfiability for decision problems). This necessitates careful maintenance of state invariants within the proof checker implementation. And thirdly, all of the above needs to be adequately optimized for practical use, whilst being formally verified.

We use a refinement-based approach to tackle each challenge in order and at the appropriate level of abstraction.

1. The verification process starts by defining an abstract, mathematical, pseudo-Boolean semantics, with respect to which the soundness of each rule is justified. For example, we prove lemmas that justify the soundness of adding two constraints and dividing a constraint by a non-zero natural number in a cutting planes proof step:

   $\vdash \text{satisfies\_npbc}\ w\ C_1 \land \text{satisfies\_npbc}\ w\ C_2 \Rightarrow$
   $\quad \text{satisfies\_npbc}\ w\ (\text{add}\ C_1\ C_2)$

   $\vdash \text{satisfies\_npbc}\ w\ C \land k \neq 0 \Rightarrow$
   $\quad \text{satisfies\_npbc}\ w\ (\text{divide}\ C\ k)$

   Here, satisfies_npbc w C says that the pseudo-Boolean constraint C is satisfied by the Boolean assignment w. We verify similar lemmas for all supported reasoning principles, the most involved of which is dominance-based strengthening. Specifically, this rule requires making a well-founded induction argument over an arbitrary user-specified order for Boolean assignments, for which we largely follow the proof from [BGMN23, Proposition 4].

2. Next, we implement a prototype proof checker that ensures that every application of a proof rule is valid, e.g., that divide is never applied with $k = 0$, throwing an error otherwise. The proof checker is verified to maintain key invariants on the proof state, especially the ones needed for dominance and optimization reasoning. Soundness of the checker is proved by induction over the sequence of proof steps. The main idea is illustrated by the following abridged lemma snippet.

   $\vdash \ldots \land \text{valid\_conf}\ \mathit{ord}\ \mathit{obj}\ \mathit{fml} \Rightarrow$
   $\quad \text{check\_step}\ \mathit{step}\ \mathit{ord}\ \mathit{obj}\ \mathit{fml}\ \ldots =$
   $\qquad \text{Some}\ (\mathit{ord}',\mathit{obj}',\mathit{fml}',\ \ldots) \Rightarrow$
   $\quad \ldots \land \text{valid\_conf}\ \mathit{ord}'\ \mathit{obj}'\ \mathit{fml}'$

   Here, valid_conf *ord obj fml* says that for any satisfying assignment w to the core constraints in formula *fml*, there exists another satisfying assignment $w' \preccurlyeq w$ which satisfies all constraints in *fml*, where $\preccurlyeq$ is the order on assignments induced by *ord* and *obj*. The lemma fragment says that, whenever checking a single proof step (check_step) succeeds and returns a new proof checker state (result Some), the valid_conf invariant is maintained for the state. Other key properties verified for check_step include showing that *fml′* and *fml* are equisatisfiable by assignments that improve the best known objective value.

3. The final phase involves refining the prototype into an optimized proof checker implementation using the CakeML tools for profiling and source code verification [MO14, GMKN17]. We manually optimize several hotspots encountered in the pseudo-Boolean proofs generated in our experimental

$$
\begin{aligned}
&\mathsf{is\_cis}\ vs\ (v_p,e_p)\ (v_t,e_t) \overset{\text{def}}{=} \\
&\quad \exists f.\ vs \subseteq \{\ 0,1,\ldots,v_p{-}1\ \} \land \mathsf{inj}\, f\ vs\ \{\ 0,1,\ldots,v_t{-}1\ \} \land \\
&\qquad \forall a\ b.\ a \in vs \land b \in vs \Rightarrow \\
&\qquad\qquad (\mathsf{is\_edge}\ e_p\ a\ b \iff \mathsf{is\_edge}\ e_t\ (f\ a)\ (f\ b)) \\
&\mathsf{connected\_subgraph}\ vs\ e \overset{\text{def}}{=} \\
&\quad \forall a\ b.\ a \in vs \land b \in vs \Rightarrow \\
&\qquad (\lambda\, x\ y.\ y \in vs \land \mathsf{is\_edge}\ e\ x\ y)^*\ a\ b \\
&\mathsf{is\_ccis}\ vs\ (v_p,e_p)\ (v_t,e_t) \overset{\text{def}}{=} \\
&\quad \mathsf{is\_cis}\ vs\ (v_p,e_p)\ (v_t,e_t) \land \mathsf{connected\_subgraph}\ vs\ e_p \\
&\mathsf{max\_ccis\_size}\ g_p\ g_t \overset{\text{def}}{=} \\
&\quad \mathsf{max}_{\mathsf{set}}\ \{\ \mathsf{card}\ vs \mid \mathsf{is\_ccis}\ vs\ g_p\ g_t\ \}
\end{aligned}
$$

$$
\begin{aligned}
&\vdash \mathsf{good\_graph}\ (v_p,e_p) \land \mathsf{good\_graph}\ (v_t,e_t) \land \\
&\quad \mathsf{encode}\ (v_p,e_p)\ (v_t,e_t) = \mathit{constraints} \Rightarrow \\
&\quad ((\exists vs.\ \mathsf{is\_ccis}\ vs\ (v_p,e_p)\ (v_t,e_t) \land \mathsf{card}\ vs = k) \iff \\
&\qquad \exists w.\ \mathsf{satisfies}\ w\ (\mathsf{set}\ \mathit{constraints}) \land \\
&\qquad\quad \mathsf{eval\_obj}\ (\mathsf{unmapped\_obj}\ v_p)\ w = v_p - k)
\end{aligned}
$$

Figure 4: *HOL definition of the size of a maximum common connected induced subgraph (MCCIS) for a pattern graph g_p and a target graph g_t (top), and a correctness theorem for encoding the MCCIS problem using PB constraints (bottom).*

evaluation, e.g., using buffered I/O to stream large proof files, and swapping to constant-time array-based constraint lookups for cutting planes steps and hash-based proof goal coverage checks in application of the dominance-based strengthening rule.

The verified proof checker backend operates most naturally and efficiently with normalized pseudo-Boolean constraints where, in addition, variables are indexed by numbers. However, this is not the most convenient interface for frontend users. Accordingly, CakePB also includes a verified pseudo-Boolean *normalizer*. As shown in Figure 3, CakePB accepts any pseudo-Boolean formula as input (normalized or otherwise) together with an externally generated kernel proof. It produces an appropriate verified conclusion about the formula, such as satisfiability status or upper and lower bounds on the objective function, depending on the type of problem and on the claims made by the proof.

3.2 Verified Graph Problem Encoders

Pseudo-Boolean formulas provide a convenient format for verified frontend encoders for graph problems, which we turn to next. Graphs are represented in HOL as a pair (v,e), where v is the number of vertices corresponding to the vertex set $\{\ 0,1,\ldots,v{-}1\ \}$, and e is an edge list representation such that is_edge $e\ a\ b$

is true iff there is an edge between vertices a and b. All graphs considered here are undirected.[1] The graph encoders use a shared graph library which formalizes these basic graph notions and provides parsing functions for standard text formats such as LAD and DIMACS.

The HOL definitions of various graph problems formalized in this paper are shown in Figures 2 and 4; we use maximum common connected induced subgraph (MCCIS) as a representative example. Given a pattern graph g_p and a target graph g_t, a subset of vertices vs of g_p is a common induced subgraph (is_cis) iff there exists an injective mapping f from vs into the target graph vertices which preserves edges and non-edges. Additionally, vs is a connected subgraph of g_p iff its vertices are pairwise connected in the reflexive transitive closure (denoted *) of the induced is_edge relation. The MCCIS size is the size of the largest common connected induced subgraph between g_p and g_t (max_ccis_size).

The MCCIS pseudo-Boolean encoding from [GMM+20, Section 3.1] is implemented as a HOL function encode. The main subtlety is connected_subgraph; briefly, connectedness is encoded using additional auxiliary variables that indicate whether a walk of length n for some $n < \min(v_p, v_t)$, exists between each pair of vertices in the chosen subgraph. The correctness theorem for encode is shown in Figure 4 (bottom). It says that a CCIS of cardinality k exists iff a satisfying assignment to the encoding *constraints* exists with objective value $v_p - k$. Therefore, minimizing the objective (unmapped_obj v_p) yields the MCCIS size. Similar theorems are proved for encodings of subgraph isomorphism and maximum clique. The value of formal verification here is twofold: to gain confidence in the pen-and-paper justification of the encodings, and to ensure that the encodings are correctly implemented in code.

3.3 End-to-End Verification

Feeding the output of each frontend encoder into CakePB yields a suite of formally verified graph proof checkers, collectively called CakePBGraph. Since we are working within the CakeML ecosystem, we can further achieve *end-to-end* verification by running the CakeML compiler on CakePBGraph to transfer the source-level correctness guarantees for the CakePBGraph checkers down to the level of their respective machine code implementations.

Let us illustrate this by briefly discussing the final correctness theorem for the maximum clique proof checker as shown in Figure 5. The assumption on Line 1 is standard for all programs written in CakeML, and states that the compiled machine code is correctly loaded in memory of an x64 machine and that the appropriate command line and file system foreign function interfaces (FFIs) are available to CakeML. The first correctness guarantee on Lines 2–4 says that the code will run without crashing and will terminate safely, possibly reporting an out-of-memory resource error. The second correctness guarantee starting at Line 5–6 says there

[1]In practice, we apply a consistency check good_graph for undirectedness and other syntactic properties when parsing input graphs. Graphs failing the check are rejected by the encoders.

```
clique_eq_str n ≝ "s VERIFIED MAX CLIQUE SIZE |CLIQUE| = " ^ toString n ^ "\n"
clique_bound_str l u ≝
 "s VERIFIED MAX CLIQUE SIZE BOUND " ^ toString l ^ " <= |CLIQUE| <= "
 ^ toString u ^ "\n"
```

```
 1 | ⊢ cake_pb_clique_run cl fs mc ms ⇒
 2 |     machine_sem mc (basis_ffi cl fs) ms ⊆
 3 |      extend_with_resource_limit
 4 |       { Terminate Success (cake_pb_clique_io_events cl fs) } ∧
 5 |     ∃out err. extract_fs fs (cake_pb_clique_io_events cl fs) =
 6 |       Some (add_stdout (add_stderr fs err) out) ∧
 7 |      (out ≠ "" ⇒
 8 |        ∃g. get_graph_dimacs fs (el 1 cl) = Some g ∧
 9 |          (length cl = 2 ∧ out = concat (print_pbf (full_encode g)) ∨
10 |           length cl = 3 ∧
11 |           (out = clique_eq_str (max_clique_size g) ∨
12 |            ∃l u. out = clique_bound_str l u ∧ (∀vs. is_clique vs g ⇒ card vs ≤ u) ∧
13 |              ∃vs. is_clique vs g ∧ l ≤ card vs)))
```

Figure 5: *End-to-end correctness theorem for CakePB with a maximum clique pseudo-Boolean encoder frontend.*

will be (possibly empty) strings *out* and *err* printed to standard output and error, respectively. The remaining lines now claim that if standard output is non-empty, then the input file was parsed in DIMACS format to a graph g (Lines 7–8), and the output is either:

- a pretty-printed pseudo-Boolean encoding of the maximum clique problem for g (Line 9), or
- a pretty-printed conclusion string which is either:
  - a verified exact maximum clique size for g formatted using clique_eq_str (Line 11), or
  - verified lower and upper bounds on clique sizes in g formatted using clique_bound_str (Lines 12–13).

Let us clarify what needs to be trusted, or at least carefully inspected, in order to claim that the conclusions by CakePBGraph checkers are formally verified:

- The HOL definitions of the graph input parsers and of various graph problems that appear in the final correctness theorems (e.g., Figure 5). We have kept these definitions as simple as possible. Notably, the internal definitions of pseudo-Boolean semantics and cutting planes used in the proof checker are *not* part of CakePBGraph's trusted base because conversion into and out of pseudo-Boolean semantics is formally verified.

- The formal HOL model of the CakeML execution environment and its correspondence with the real system on which CakePBGraph runs. CakeML has been used in various other proof checkers, e.g., by [THM23], and its target architecture models have been validated extensively [TMK+19].
- The HOL4 theorem prover, including its logic, implementation and execution environment. The prover follows an LCF-style design [SN08] with a well-separated and trustworthy kernel responsible for checking every logical inference.

A trusted base for *binary code extraction* [KMTM18] as above is of the highest assurance standard for formally verified software—correctness is proved within a single system down to the machine code that runs. This provides a gold standard of trustworthiness for subgraph solving, in contrast to prior unverified proof checking approaches.

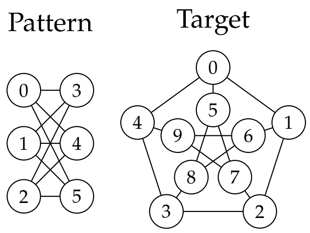
4 Proof Elaboration

CakePBGraph verification helps solver *users* who wish to attain a high level of trust in solver conclusions. In this section, we discuss our new *elaboration* phase, which aids solver *authors* who wish to add trustworthy proof logging and checking to their tools.

The convenience afforded by proof elaboration is illustrated in the workflow in Figure 1. First, solver authors can design their proof output with respect to their own (untrusted) pseudo-Boolean encodings, without following the verified encodings from CakePBGraph exactly; elaboration helps to automatically line up (where possible) untrusted and verified encodings. Second, elaboration supports an *augmented* proof format with syntactic sugar that makes proof logging much easier at runtime; elaboration then fills in the necessary details to convert the proof into the kernel format understood by CakePBGraph. The VeriPB proof elaborator also performs (unverified) proof checking during the translation process, helping solver authors to detect bugs in their proof logging or solver code even before the formal verification process starts.

4.1 Lining up Encodings

Many VeriPB proof rules refer to constraints by positive integer *constraint IDs*, assigned automatically in order of appearance in the proof. It would be quite a hassle for solver authors to keep track of the exact order in which constraints in the encoding are generated by CakePBGraph. Fortunately, it is straightforward to instead recover an ID by rederiving the constraint, which provides it with a new, known ID, before it is used. This can either be done upfront, at the start of the proof, or lazily (which avoids a potentially large overhead for instances with very short proofs). A useful fact is that the two constraints do not need to match



Verified Encoding

```
min: 1 x0_n 1 x1_n 1 x2_n 1 x3_n 1 x4_n 1 x5_n ;
1 x0_n 1 x0_0 1 x0_1 1 x0_2 1 x0_3 1 x0_4 \
  1 x0_5 1 x0_6 1 x0_7 1 x0_8 1 x0_9 = 1 ;
1 x1_n 1 x1_0 1 x1_1 1 x1_2 1 x1_3 1 x1_4 \
  1 x1_5 1 x1_6 1 x1_7 1 x1_8 1 x1_9 = 1 ;
... 1172 omitted constraints ...
```

Augmented Proof

```
pseudo-Boolean proof version 2.0
...
* Specifying a partial solution
soli x5_9 x2_7 ...
(58 omitted literals) ...
* Unit propagation step
u 1 ~x4_0 >= 1 ;



...
conclusion BOUNDS 2 2
end pseudo-Boolean proof
```

Kernel Proof

```
pseudo-Boolean proof version 2.0
...
* Specifying a full solution
soli x0_n x1_n ...
(304 omitted literals) ...
* Derivation by cutting planes
red 1 ~x4_0 >= 1 ; ; begin
  pol 8784 8778 + 8772 + 8766 + \
    ... 8133 13 * + 8085 13 * +
end 8786
...
conclusion BOUNDS 2 : 8798 2
end pseudo-Boolean proof
```

Figure 6: *(Top) MCCIS problem encoding for the pattern graph $K_{3,3}$ and the target Petersen graph. (Bottom) An augmented proof generated by a solver on the left, and a corresponding elaborated kernel proof on the right; kernel annotations in **bold**. When run on the kernel proof, CakePBGraph outputs:* `s VERIFIED MAX CCIS SIZE |CCIS| = 4`*. This corresponds to the conclusion in the proof, which claims that at least two of the six pattern vertices must be mapped to null.*

exactly—it is sufficient that they are close enough so that VeriPB can automatically check and prove that one of them follows from the other.

When it comes to variable names, the solver proof logging routines are required to agree exactly with the CakePBGraph encoding. This is an easier task, however, since VeriPB and CakePB both support expressive variable names. For example, for subgraph mapping problems, we use the protocol that the variable name `x1_2` means that pattern vertex 1 will be mapped to target vertex 2.

4.2 Elaborating on Syntactic Sugar

The augmented proof format contains a number of rules designed to support the ease of proof logging. Chief among these is *reverse unit propagation (RUP)*, which allows to add a constraint when the VeriPB proof checker can easily verify that it is implied by applying unit propagation. Such RUP steps occur frequently in proofs in many applications, and so have to be dealt with efficiently by the proof checker, but implementing efficient formally verified unit propagation is a challenging task even for the simpler case of CNF [FBL18]. Instead, a RUP rule application deriving C from F is converted to an explicit cutting planes proof of contradiction from $F \cup \{\neg C\}$. This is possible since unit propagation on the latter set of constraints leads to a violation (by the definition of RUP), and this in turn means that pseudo-Boolean conflict analysis can be used to derive contradiction. This algorithm is more involved than CNF-based conflict analysis as used in SAT solvers, but we employ a procedure similar to the PB conflict analysis in [EN18] for this. For optimization problems, the augmented format allows incumbent solutions to be partially specified, so long as the given assignment unit propagates to a full solution; the kernel format will always specify a full solution instead. This is illustrated in Figure 6.

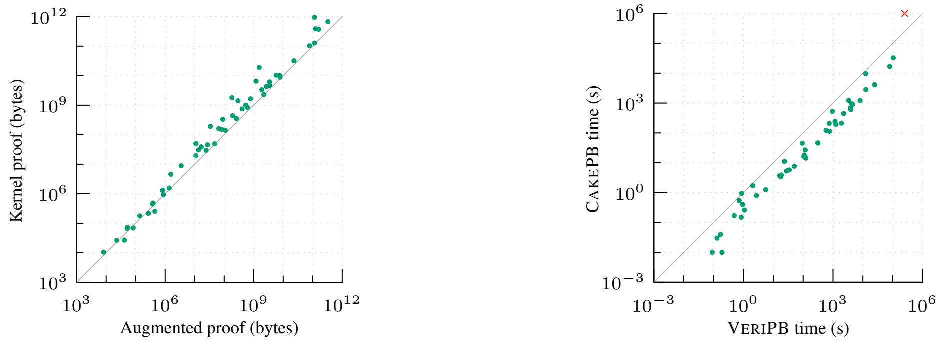
Another convenient rule is *syntactic implication*, where a constraint to be derived is implied by a single (unspecified) previous constraint by simple syntactic manipulations. This condition is again easy to check, but the elaborator converts this into an explicit derivation or explicitly annotates the kernel proof with IDs. Yet another important aspect that we are ignoring here, but which is crucial for efficient proof checking, is deletion of constraints no longer needed in the proof.

Finally, applications of strengthening rules generate a separate proof goal for each constraint currently in use in the proof, which is a potentially huge overhead, but often most of these proof goals are obvious and can be skipped in the augmented format (e.g., if they can be obtained by RUP or syntactic implication). The proof elaborator fills in the necessary missing details for such proof goals.

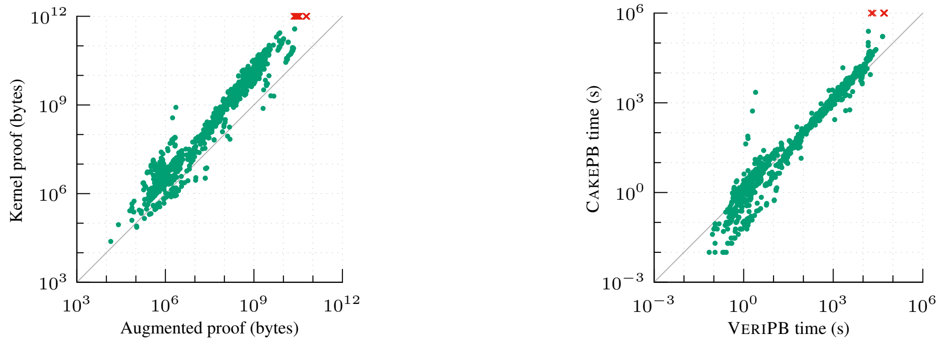
5 Experiments

To validate our approach, we performed experiments on a cluster of machines with dual AMD EPYC 7643 processors, 2TBytes RAM, and a RAID array of solid state drives, running Ubuntu 22.04. We ran up to 40 jobs in parallel, and limited each individual process to 64GBytes RAM. Note that performance of the verification process is strongly affected by I/O and memory cache speeds, and so we do not expect running time measurements to be highly reproducible, but they should still be indicative of the feasibility of the approach and the slowdowns that one might encounter. We used the Glasgow Subgraph Solver [MPT20] as the proof-producing solver for all experiments, and made small modifications so that it would lazily recover constraint IDs as required. The results are plotted on an instance by instance basis in Figure 7 and explained below.

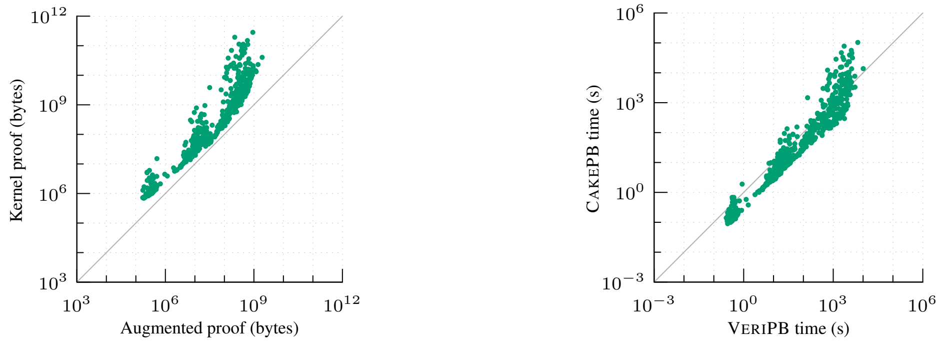
For maximum clique, we took the 54 instances from the Second DIMACS Implementation Challenge [JT96] that [GMM+20] were able to check. We managed to produce proofs for and formally verify 50 of these instances; for the 4 instances that we could not verify, 3 were due to VeriPB taking over one week to check the proof files, and the final one to the 64GByte memory limit for the verified



(a) *Max clique*



(b) *Subgraph isomorphism*



(c) *Max common connected induced subgraph*

Figure 7: *Experiments using the Glasgow Subgraph Solver on (a) max clique, (b) subgraph isomorphism, and (c) max common connected induced subgraph problem instances. In the left column, comparisons of kernel and augmented proof sizes; in the right column, time comparisons for verified and unverified checking of kernel and augmented proofs, respectively. Crosses indicate failures due to space or memory limits.*

checker. Over the successfully checked instances, translating augmented proofs to kernel proofs took, on average, 18% longer than simply verifying the proofs, and produced proof files that were on average 2.26 times as large. However, verified checking of these kernel proofs was consistently faster than checking the original augmented proofs using VeriPB: the average running time was 3.8 times lower.

For subgraph isomorphism, we used the same subset of 1,226 small-to-medium-sized instances from the benchmark set in [KMS16] as was studied by [GMN20]. We were able to verify 417 satisfiable and 784 unsatisfiable instances; 13 instances failed due to memory limits on the verified checker, and 12 instances when the converted kernel proofs exceeded 500GBytes in size. Performance-wise, running VeriPB and asking it to output a kernel proof was on average 27% slower than verification alone. Producing the verified encoding was never a significant cost in the process. Verifying kernel proofs was on average 2.4 times slower than verifying the original, augmented proofs; the former were on average 10.5 times larger than the latter.

For maximum common connected induced subgraph, we used a database of randomly generated instances [CFV07, DFSV03], and ran the solver in clique reformulation mode. We were able to verify all 690 instances involving up to 20 vertices in each graph. Elaborating the proofs took on average 43% longer than verifying them using VeriPB, and the proofs were on average 14.7 times larger. However, verifying the kernel proofs using CakePB took on average only 9% longer than using VeriPB for the original, augmented proofs.

Across each problem family, producing formally verified encodings was always extremely cheap, and asking VeriPB to produce an elaborated kernel proof was never substantially more expensive than simply checking the augmented proof. This is to be expected: VeriPB already has to produce nearly all of the information needed for proof elaboration to check a proof anyway. Checking elaborated proofs was sometimes a little faster than checking the original, augmented proof, and sometimes a little slower, and we were able to formally check almost every proof that was amenable to unverified checking.

6 Conclusion

In this paper, we present the first efficient toolchain for formal end-to-end verification of state-of-the-art subgraph solving. Our design is easily adaptable, which opens up the possibility of bringing formal verification to other combinatorial problem domains where problem instances can be suitably represented using the expressivity of 0–1 integer linear programs. In fact, our formally verified CakePB proof checker equipped with a CNF frontend has also been used for SAT solving in the SAT Competition 2023 [BMM+23], supporting, also for the first time, efficient verified proof logging and checking for the full range of advanced techniques used in modern SAT solvers such as cardinality reasoning, Gaussian elimination, and symmetry breaking. A future challenge of particular interest would be to provide a formally verified setting for the proof logging techniques for constraint programming developed in a sequence of papers by [EGMN20, GMN22] and [MM23]. It would also be valuable to expand the reach of pseudo-Boolean proof logging to problems like (projected) model enumeration problems, which were dealt with in a somewhat ad-hoc fashion by [GMM+20].

To further improve performance, it would be highly desirable to enhance the VeriPB elaborator with *proof trimming* to be able to remove unnecessary proof steps before handing the kernel proof to CakePB. Currently, our system verifies all of the steps carried out by the solver to reach its conclusion. This is useful for detecting solver bugs, but for storing and distributing proofs a trimmed proof would suffice and could be much faster to verify. Another significant source of performance gains could come from switching from a text proof format to a binary format: although this would lose some human-readability, our experiments suggest that text parsing often forms a substantial portion of the elaboration and checking times.

Acknowledgements

Stephan Gocht and Jakob Nordström were supported by the Swedish Research Council grant 2016-00782, and Jakob Nordström also received funding from the Independent Research Fund Denmark grant 9040-00389B. Ciaran McCreesh was supported by a Royal Academy of Engineering research fellowship, and by the Engineering and Physical Sciences Research Council [grant number EP/X030032/1]. Magnus Myreen was supported by Swedish Research Council grant 2021-05165. Andy Oertel was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Yong Kiam Tan was supported by A*STAR, Singapore. Part of this work was carried out while taking part in the Dagstuhl workshops 22411 "Theory and Practice of SAT and Combinatorial Solving" and 23261 "SAT Encodings and Beyond", as well as in the extended reunion of the program "Satisfiability: Theory, Practice, and Beyond" in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley.

References

[AGJ+18] 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.

[BDM23] Milan Bankovic, Ivan Drecun, and Filip Maric. A proof system for graph (non)-isomorphism verification. *Logical Methods in Computer Science*, 19(1), February 2023.

[BGMN23] 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*.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[BMM$^+$23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BMN22] Bart Bogaerts, Ciaran McCreesh, and Jakob Nordström. Solving with provably correct results: Beyond satisfiability, and towards constraint programming. Tutorial at the *28th International Conference on Principles and Practice of Constraint Programming*. Slides available at `http://www.jakobnordstrom.se/presentations/`, August 2022.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[BR07] Robert Bixby and Edward Rothberg. 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, February 2007.

[CAH23] Cayden R. Codel, Jeremy Avigad, and Marijn J. H. Heule. Verified encodings for SAT solvers. In *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD '23)*, pages 141–151, October 2023.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, January 2007.

[CHH$^+$17] 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.

[CKSW13] William Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A hybrid branch-and-bound approach for exact rational mixed-integer programming. *Mathematical Programming Computation*, 5(3):305–344, September 2013.

[CMS17] 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.

[CMS19] Luís Cruz-Filipe, João P. Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the Boolean Pythagorean triples problem. *Journal of Automated Reasoning*, 63(3):695–722, October 2019.

[DFSV03] Massimo De Santo, Pasquale Foggia, Carlo Sansone, and Mario Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, May 2003.

[EGMN20] 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.

[EN18] 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.

[FBL18] Mathias Fleury, Jasmin Christian Blanchette, and Peter Lammich. A verified SAT solver with watched literals using imperative HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*, pages 158—171, January 2018.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM^{+}20] 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.

[GMM^{+}23] Stephan Gocht, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. End-to-end verification for subgraph solving: Supplementary material. `https://doi.org/10.5281/zenodo.10369401`, December 2023.

[GMN20] 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.

[GMN22] 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.

[GN21] 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.

[GSD19] 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.

[GSVW14] Maria Garcia de la Banda, Peter J. Stuckey, Pascal Van Hentenryck, and Mark Wallace. The future of optimization technology. *Constraints*, 19(2):126–138, April 2014.

[HHW13a] 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.

[HHW13b] 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.

[HK17] Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Communications of the ACM*, 60(8):70–79, August 2017.

[JT96] David S. Johnson and Michael A. Trick. Introduction to the second DIMACS challenge: Cliques, coloring, and satisfiability. In *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge,* volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–10. American Mathematical Society, 1996.

[KMS16] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In *10th International Conference on Learning and Intelligent Optimization (LION '16), Selected Revised Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, May-June 2016.

[KMTM18] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *Proceedings of the 9th International Conference on Interactive Theorem Proving (ITP '18)*, volume 10895 of *Lecture Notes in Computer Science*, pages 362–369. Springer, July 2018.

[Lam20] 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.

[MM23] 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.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[MPT20] 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.

[SFL$^+$21] Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. CoqQFBV: A scalable certified SMT quantifier-free bit-vector solver. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV '21)*, volume 12760 of *Lecture Notes in Computer Science*, pages 149–171. Springer, July 2021.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, August 2008.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[TMK$^+$19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[WHH14] 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.

Certified MaxSAT Preprocessing

Abstract

Building on the progress in Boolean satisfiability (SAT) solving over the last decades, maximum satisfiability (MaxSAT) has become a viable approach for solving NP-hard optimization problems. However, ensuring correctness of MaxSAT solvers has remained a considerable concern. For SAT, this is largely a solved problem thanks to the use of proof logging, meaning that solvers emit machine-verifiable proofs to certify correctness. However, for MaxSAT, proof logging solvers have started being developed only very recently. Moreover, these nascent efforts have only targeted the core solving process, ignoring the preprocessing phase where input problem instances can be substantially reformulated before being passed on to the solver proper.

In this work, we demonstrate how pseudo-Boolean proof logging can be used to certify the correctness of a wide range of modern MaxSAT preprocessing techniques. By combining and extending the VeriPB and CakePB tools, we provide formally verified end-to-end proof checking that the input and preprocessed output MaxSAT problem instances have the same optimal value. An extensive evaluation on applied MaxSAT benchmarks shows that our approach is feasible in practice.

1 Introduction

The development of Boolean satisfiability (SAT) solvers is arguably one of the true success stories of modern computer science—today, SAT solvers are routinely used as core engines in many types of complex automated reasoning systems. One example of this is SAT-based optimization, usually referred to as *maximum satisfiability (MaxSAT) solving.* The improved performance of SAT solvers, coupled

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.

with increasingly sophisticated techniques for using SAT solver calls to reason about optimization problems, have made MaxSAT solvers a powerful tool for tackling real-world NP-hard optimization problems [BHvMW21].

However, Modern MaxSAT solvers are quite intricate pieces of software, and it has been shown repeatedly in the MaxSAT evaluations [Maxb] that even the best solvers sometimes report incorrect results. This was previously a serious issue also for SAT solvers (see, e.g., [BLB10]), but the SAT community has essentially eliminated this problem by requiring that solvers should be *certifying* [ABM+11, MMNS11], i.e., not only report whether a given formula is satisfiable or unsatisfiable but also produce a machine-verifiable proof that this conclusion is correct. Many different SAT proof formats such as RUP [GN03], TraceCheck [Bie06], GRIT [CMS17], and LRAT [CHH+17] have been proposed, with DRAT [HHW13a, HHW13b, WHH14] established as the de facto standard; for the last ten years, proof logging has been compulsory in the (main track of the) SAT competitions [SAT]. It is all the more striking, then, that until recently no similar developments have been observed in MaxSAT solving.

1.1 Previous Work

A first natural question to ask—since MaxSAT solvers are based on repeated calls to SAT solvers—is why we cannot simply use SAT proof logging also for MaxSAT. The problem is that DRAT can only reason about clauses, whereas MaxSAT solvers argue about costs of solutions and values of objective functions. Translating such claims to clausal form would require an external tool to certify correctness of the translation. Also, such clausal translations incur a significant overhead and do not seem well-adapted for, e.g., counting arguments in MaxSAT.

While there have been several attempts to design proof systems specifically for MaxSAT solving [BLM07, FMSV20, IBJ22, LNOR11, MIB+19, MM11, PCH20, PCH21, PCH22], none of these have come close to providing a general proof logging solution, because they apply only for very specific algorithm implementations and/or fail to capture the full range of techniques used. Recent papers have instead proposed using pseudo-Boolean proof logging with VeriPB [BGMN23, GN21] to certify correctness of so-called solution-improving solvers [VDB22] and core-guided solvers [BBN+23]. Although these works demonstrate, for the first time, practical proof logging for modern MaxSAT solving, the methods developed thus far only apply to the core solving process. This ignores the preprocessing phase, where the input formula can undergo major reformulation. State-of-the-art solvers sometimes use stand-alone preprocessor tools, or sometimes integrate preprocessing-style reasoning more tightly within the MaxSAT solver engine, to speed up the search for optimal solutions. Some of these preprocessing techniques are lifted from SAT to MaxSAT, but there are also native MaxSAT preprocessing methods that lack analogies in SAT solving.

1.2 Our Contribution

In this paper, we show, for the first time, how to use pseudo-Boolean proof logging with VeriPB to produce proofs of correctness for a wide range of preprocessing techniques used in modern MaxSAT solvers. VeriPB proof logging has previously been successfully used not only for core MaxSAT search as discussed above, but also for advanced SAT solving techniques (including symmetry breaking) [BGMN23, GMNO22, GN21], subgraph solving [GMM+20, GMM+24, GMN20], constraint programming [EGMN20, GMN22, MM23, MMN24], and 0–1 ILP presolving [HOGN24], and we add MaxSAT preprocessing to this list.

In order to do so, we extend the VeriPB proof format to include an *output section* where a reformulated output can be presented, and where the pseudo-Boolean proof establishes that this output formula and the input formula are *equioptimal*, i.e., have optimal solutions of the same value. We also enhance CakePB [BMM+23, GMM+24]—a verified proof checker for pseudo-Boolean proofs—to handle proofs of reformulation. In this way, we obtain an end-to-end formally verified toolchain for certified preprocessing of MaxSAT instances.

It is worth noting that although preprocessing is also a critical component in SAT solving, we are not aware of any tool for certifying reformulations even for the restricted case of decision problems, i.e., showing that formulas are *equisatisfiable*—the DRAT format and tools support proofs that satisfiability of an input CNF formula F implies satisfiability of an output CNF formula G but not the converse direction (except in the special case where F is a subset of G). To the best of our knowledge, our work presents the first practical tool for proving (two-way) equisatisfiability or equioptimality of reformulated problems.

We have performed computational experiments running a MaxSAT preprocessor with proof logging and proof checking on benchmarks from the MaxSAT evaluations [Maxb]. Although there is certainly room for improvements in performance, these experiments provide empirical evidence for the feasibility of certified preprocessing for real-world MaxSAT benchmarks.

1.3 Organization of This Paper

After reviewing preliminaries in Section 2, we explain our pseudo-Boolean proof logging for MaxSAT preprocessing in Section 3, and Section 4 discusses verified proof checking. We present results from a computational evaluation in Section 5, after which we conclude with a summary and outlook for future work in Section 6.

2 Preliminaries

We write ℓ to denote a literal, i.e., a $\{0,1\}$-valued Boolean variable x or its negation $\overline{x} = 1 - x$. A *clause* $C = \ell_1 \vee \ldots \vee \ell_k$ is a disjunction of literals, where a *unit clause* consists of only one literal. A formula in *conjunctive normal form (CNF)*

$F = C_1 \wedge \ldots \wedge C_m$ is a conjunction of clauses, where we think of clauses and formulas as sets so that there are no repetitions and order is irrelevant.

A *pseudo-Boolean (PB) constraint* is a 0–1 linear inequality $\sum_j a_j \ell_j \geq b$, where, when convenient, we can assume all literals ℓ_j to refer to distinct variables and all integers a_j and b to be positive (so-called *normalized form*). A *pseudo-Boolean formula* is a conjunction of such constraints. We identify the clause $C = \ell_1 \vee \cdots \vee \ell_k$ with the pseudo-Boolean constraint $\mathrm{PB}(C) = \ell_1 + \cdots + \ell_k \geq 1$, so a CNF formula F is just a special type of PB formula $\mathrm{PB}(F) = \{\mathrm{PB}(C) \mid C \in F\}$.

A *(partial) assignment* ρ mapping variables to $\{0, 1\}$, is extended to literals by respecting the meaning of negation, satisfies a PB constraint $\sum_j a_j \ell_j \geq b$ if $\sum_{\ell_j : \rho(\ell_j)=1} a_j \geq b$ (assuming normalized form). A PB formula is satisfied by ρ if all constraints in it are. We also refer to total satisfying assignments ρ as *solutions*. In a *pseudo-Boolean optimization (PBO)* problem we ask for a solution minimizing a given *objective* function $O = \sum_j c_j \ell_j + W$, where c_j and W are integers and W represents a trivial lower bound on the minimum cost.

2.1 Pseudo-Boolean Proof Logging Using Cutting Planes

The pseudo-Boolean proof logging in VeriPB is based on the *cutting planes* proof system [CCT87] with extensions as discussed briefly next. We refer the reader to [BN21] for and in-depth discussion of cutting planes and to [BGMN23, Goc22, HOGN24, Ver] for more detailed information about the VeriPB proof system and format.

A pseudo-Boolean proof maintains two sets of *core constraints* $\mathcal{C}$ and *derived constraints* $\mathcal{D}$ under which the objective O should be minimized. At the start of the proof, $\mathcal{C}$ is initialized to the constraints in the input formula F. Any constraints derived by the rules described below are placed in $\mathcal{D}$, from where they can later be moved to $\mathcal{C}$ (but not vice versa). The proof system semantics preserves the invariant that the optimal value of any solution to $\mathcal{C}$ and to the original input problem F is the same. New constraints can be derived from $\mathcal{C} \cup \mathcal{D}$ by performing *addition* of two constraints or *multiplication* of a constraint by a positive integer, and *literal axioms* $\ell \geq 0$ can be used at any time. Additionally, we can apply *division* to $\sum_j a_j \ell_j \geq b$ by a positive integer d followed by rounding up to obtain $\sum_j \lceil a_j / d \rceil \ell_j \geq \lceil b/d \rceil$, and *saturation* to yield $\sum_j \min\{a_j, b\} \cdot \ell_j \geq b$ (where we again assume normalized form).

The negation of a constraint $C = \sum_j a_j \ell_j \geq b$ is $\neg C = \sum_j a_j \ell_j \leq b - 1$. For a (partial) assignment ρ we write $C\!\upharpoonright_\rho$ for the *restricted constraint* obtained by replacing literals in C assigned by ρ with their values and simplifying. We say that C *unit propagates* ℓ *under* ρ if $C\!\upharpoonright_\rho$ cannot be satisfied unless ℓ is assigned to 1. If repeated unit propagation on all constraints in $\mathcal{C} \cup \mathcal{D} \cup \{\neg C\}$, starting with the empty assignment $\rho = \emptyset$, leads to contradiction in the form of an unsatisfiable constraint, we say that C follows by *reverse unit propagation (RUP)* from $\mathcal{C} \cup \mathcal{D}$. Such (efficiently verifiable) RUP steps are allowed in VeriPB proofs as a convenient way to avoid writing out an explicit cutting planes derivation. We use the same notation $C{\restriction}_\omega$ to denote the result of applying to C a *(partial) substitution* ω, which can map variables not only to $\{0,1\}$ but also to literals, and extend this notation to sets of constraints by taking unions.

In addition to the above rules, which derive semantically implied constraints, there is a *redundance-based strengthening rule*, or just *redundance rule* for short, that can derive non-implied constraints C as long as they do not change the feasibility or optimal value. This can be guaranteed by exhibiting a *witness substitution* ω such that for any total assignment α satisfying $\mathcal{C} \cup \mathcal{D}$ but violating C, the composition $\alpha \circ \omega$ is another total assignment that satisfies $\mathcal{C} \cup \mathcal{D} \cup \{C\}$ and yields an objective value that is at least as good. Formally, C can be derived from $\mathcal{C} \cup \mathcal{D}$ by exhibiting ω and subproofs for

$$\mathcal{C} \cup \mathcal{D} \cup \{\neg C\} \vdash (\mathcal{C} \cup \mathcal{D} \cup \{C\}){\restriction}_\omega \cup \{O \geq O{\restriction}_\omega\}\,, \tag{1}$$

using the previously discussed rules (where the notation $\mathcal{C}_1 \vdash \mathcal{C}_2$ means that the constraints $\mathcal{C}_2$ can be derived from the constraints $\mathcal{C}_1$).

During preprocessing, constraints in the input formula are often deleted or replaced by other constraints, in which case the proof should establish that these deletions maintain equioptimality. Removing constraints from the derived set $\mathcal{D}$ is unproblematic, but unrestricted deletion from the core set $\mathcal{C}$ can clearly introduce spurious better solutions. Therefore, removing C from $\mathcal{C}$ can only be done by the *checked deletion rule*, which requires a proof that the redundance rule can be used to rederive C from $\mathcal{C} \setminus \{C\}$ (see [BGMN23] for a more detailed explanation).

Finally, it turns out to be useful to allow replacing O by a new objective O' using an *objective function update rule*, as long as this does not change the optimal value of the problem. Formally, updating the objective from O to O' requires derivations of the two constraints $O \geq O'$ and $O' \geq O$ from the core set $\mathcal{C}$, which shows that any satisfying solution to $\mathcal{C}$ has the same value for both objectives. More details on this rule can be found in [HOGN24].

2.2 Maximum Satisfiability

A WCNF instance of (weighted partial) maximum satisfiability $\mathcal{F}^W = (F_H, F_S)$ is a conjunction of two CNF formulas F_H and F_S with *hard* and *soft* clauses, respectively, where soft clauses $C \in F_S$ have positive weights w^C. A solution ρ to $\mathcal{F}^W$ must satisfy F_H and has value $\textsc{cost}(F_S, \rho)$ equal to the sum of weights of all soft clauses not satisfied by ρ. The optimum $\textsc{opt}(\mathcal{F}^W)$ of $\mathcal{F}^W$ is the minimum of $\textsc{cost}(F_S, \rho)$ over all solutions ρ, or ∞ if no solution exists.

State-of-the-art MaxSAT preprocessors such as MaxPre [IBJ22, KBSJ17] take a slightly different *objective-centric* view [BJ19] of MaxSAT instances $\mathcal{F} = (F, O)$ as consisting of a CNF formula F and an objective function $O = \sum_j c_j \ell_j + W$ to be minimized under assignments ρ satisfying F. A WCNF MaxSAT instance $\mathcal{F}^W = (F_H, F_S)$ is converted into objective-centric form $\textsc{ObjMaxSAT}(\mathcal{F}^W) = (F, O)$ by letting the formula $F = F_H \cup \{C \vee b_C \mid C \in F_S, |C| > 1\}$ of $\textsc{ObjMaxSAT}(\mathcal{F}^W)$ consist of the hard clauses of $\mathcal{F}^W$ and the non-unit soft clauses in F_S, each extended with a fresh variable b_C that does not appear in any other clause. The objective $O = \sum_{(\bar{\ell}) \in F_S} w^{(\bar{\ell})} \ell + \sum w^C b_C$ contains literals ℓ for all unit soft clauses $\bar{\ell}$ in F_S as well as literals for all new variables b_C, with coefficients equal to the weights of the corresponding soft clauses. In other words, each unit soft clause $\bar{\ell} \in F_S$ of weight w is transformed into the term $w \cdot \ell$ in the objective function O, and each non-unit soft clause C is transformed into the hard clause $C \vee b_C$ paired with the unit soft clause $(\bar{b}_C)$ with same weight as C. The following observation summarizes the properties of ObjMaxSAT($\mathcal{F}^W$) that are central to our work.

Observation 1. *For any solution ρ to a WCNF MaxSAT instance $\mathcal{F}^W$ there exists a solution ρ' to $(F, O) =$ ObjMaxSAT($\mathcal{F}^W$) with $O(\rho') = \textsc{cost}(\mathcal{F}^W, \rho)$. Conversely, if ρ' is a solution to ObjMaxSAT($\mathcal{F}^W$), then there exists a solution ρ of $\mathcal{F}^W$ for which $\textsc{cost}(\mathcal{F}^W, \rho) \leq O(\rho')$.*

For the second part of the observation, the reason $O(\rho')$ is only an upper bound on $\textsc{cost}(\mathcal{F}^W, \rho)$ is that the encoding forces b_C to be true whenever C is not satisfied by an assignment but not vice versa.

An objective-centric MaxSAT instance (F, O), in turn, clearly has the same optimum as the pseudo-Boolean optimization problem of minimizing O subject to PB(F). For the end-to-end formal verification, the fact that this coincides with $\textsc{opt}(\mathcal{F}^W)$ needs to be formalized into theorems as shown in Figure 4.

3 Proof Logging for MaxSAT Preprocessing

We now discuss how pseudo-Boolean proof logging can be used to reason about correctness of MaxSAT preprocessing steps. Our approach maintains the invariant that the current working instance in the preprocessor is synchronized with the PB constraints in the core set $\mathcal{C}$ as described in Section 2.2. At the end of each preprocessing step (i.e., application of a preprocessing technique) the set of derived constraints $\mathcal{D}$ is empty. All constraints derived in the proof as described in this section are moved to the core set, and constraints are always removed by checked deletion from the core set. Full technical details are in Appendix A.

3.1 Overview

All our preprocessing steps maintain *equioptimality*, which means that if preprocessing of the WCNF MaxSAT instance $\mathcal{F}^W$ yields the output instance $\mathcal{F}_P^W$, then the equality $\textsc{opt}(\mathcal{F}^W) = \textsc{opt}(\mathcal{F}_P^W)$ is guaranteed to hold. Our preprocessing is *certified*, meaning that we provide a machine-verifiable proof justifying this claimed equality. Our discussion below focuses on input instances that have solutions, but our techniques also handle the—arguably less interesting—case of $\mathcal{F}^W$ not having solutions; details are in Appendix A.5.

An overview of the workflow of our certifying MaxSAT preprocessor is shown in Figure 1. Given a WCNF instance $\mathcal{F}^W$ as input, the preprocessor proceeds in five

	preprocessing (MaxSAT)	*proof (pseudo-Boolean)*
1. Initialization	$(\mathcal{F}^W, 0)$	$(\mathrm{PB}(F^0), O^0)$ where $(F^0, O^0) = \textsc{ObjMaxSAT}(\mathcal{F}^W)$
2. Preprocessing on WCNF	$(\mathcal{F}_1^W, \textsc{lb}^1)$	(C^1, O^1)
3. Conversion to objective-centric	$(F^2, O^2 + \textsc{lb}^1)$ where $(F^2, O^2) = \textsc{ObjMaxSAT}(\mathcal{F}_1^W)$	$(\mathrm{PB}(F^2), O^2 + \textsc{lb}^1)$
4. Preprocessing on objective-centric	(F^3, O^3)	$(\mathrm{PB}(F^3), O^3)$
5. Constant removal	(F^4, O^4) where $F^4 = F^3 \wedge (b^{W^3})$ $O^4 = O^3 - W^3 + W^3 b^{W^3}$	$(\mathrm{PB}(F^4), O^4)$
Output	Preprocessed WCNF $\mathcal{F}_P^W = (F^4, F_S^P)$	Proof of equioptimality of $\mathrm{PB}(F^0)$ under O^0 and $\mathrm{PB}(F^4)$ under O^4

Figure 1: *Overview of the five stages of certified MaxSAT preprocessing of a WCNF instance $\mathcal{F}^W$. The middle column contains the state of the working MaxSAT instance as a WCNF instance and a lower bound on its optimum cost (Stages 1–2), or as an objective-centric instance (Stages 3–5). The right column contains a tuple (C, O) with the set C of core constraints, and objective O, respectively, of the proof after each stage.*

stages (illustrated on the left in Figure 1), and then outputs a preprocessed MaxSAT instance $\mathcal{F}_P^W$ together with a pseudo-Boolean proof that $\textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}^W)) = \textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}_P^W))$. For certified MaxSAT preprocessing, this proof can then be fed to a formally verified checker as in Section 4 to verify that (a) the initial core constraints in the proof correspond exactly to the clauses in $\textsc{ObjMaxSAT}(\mathcal{F}^W)$, (b) each step in the proof is valid, and (c) the final core constraints in the proof correspond exactly to the clauses in $\textsc{ObjMaxSAT}(\mathcal{F}_P^W)$. Below, we provide more details on the five stages of the preprocessing flow.

Stage 1: Initialization.

An input WCNF instance $\mathcal{F}^W$ is transformed to pseudo-Boolean format by converting it to an objective-centric representation $(F^0, O^0) = \textsc{ObjMaxSAT}(\mathcal{F}^W)$ and then representing all clauses in F^0 as pseudo-Boolean constraints as described in Section 2.2. The VeriPB proof starts out with core constraints $\mathrm{PB}(F^0)$ and objective O^0. The preprocessor maintains a lower bound on the optimal cost of the working instance, which is initialized to 0 for the input $\mathcal{F}^W$.

Stage 2: Preprocessing on the Initial WCNF Representation.

During preprocessing on the WCNF representation, a (very limited) set of simplification techniques are applied on the working formula. At this stage the preprocessor removes duplicate, tautological, and blocked clauses [JBH10]. Additionally, hard unit clauses are unit propagated and clauses subsumed by hard clauses are removed. Importantly, the preprocessor is performing these simplifications on a WCNF MaxSAT instance where it deals with hard and soft clauses. As the pseudo-Boolean proof has no concept of hard or soft clauses, the reformulation steps must be expressed in terms of the constraints in the proof. The next example illustrates how reasoning with different types of clauses is logged in the proof.

Example 1. Suppose the working instance has two duplicate clauses C and D. If both are hard, then the proof has two identical constraints $\mathrm{PB}(C)$ and $\mathrm{PB}(D)$ in the core set, and $\mathrm{PB}(D)$ can be deleted since it follows from $\mathrm{PB}(C)$ by reverse unit propagation (RUP). If D is instead a non-unit soft clause, the proof has the constraint $\mathrm{PB}(D \vee b_D)$ and the term $w^D b_D$ in the objective, where b_D does not appear in any other constraint. Then in the proof we (1) remove the RUP constraint $\mathrm{PB}(D \vee b_D)$, (2) introduce $\overline{b}_D \geq 1$ by redundance-based strengthening using the witness $\{b_D \rightarrow 0\}$, (3) remove the term $w^D b_D$ from the objective, and (4) delete $\overline{b}_D \geq 1$ with the witness $\{b_D \rightarrow 0\}$.

Stage 3: Conversion to Objective-Centric Representation.

In order to apply more simplification rules in a cost-preserving way, the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ at the end of Stage 2 is converted into the corresponding objective-centric representation that takes the lower-bound LB inferred during Stage 1 into account. More specifically, the preprocessor next converts its working MaxSAT instance into the objective-centric instance $\mathcal{F}_2 = (F^2, O^2 + \textsc{lb})$ where $(F^2, O^2) = \textsc{ObjMaxSAT}(\mathcal{F}_1^W)$.

Here it is important to note that at the end of Stage 2, the core constraints $\mathcal{C}^1$ and objective O^1 of the proof are not necessarily $\mathrm{PB}(F^2)$ and $O^2 + \textsc{lb}$, respectively. Specifically, consider a unit soft clause $(\overline{\ell})$ of $\mathcal{F}_1^W$ obtained by shrinking a non-unit soft clause $C \supseteq (\overline{\ell})$ of the input instance, with weight w^C. Then the objective function O^2 in the preprocessor will include the term $w^C \ell$ that does not appear in the objective function O^1 in the proof. Instead, O^1 contains the term $w^C b_C$ and $\mathcal{C}^1$ the constraint $\overline{\ell} + b_C \geq 1$ where b_C is the fresh variable added to C in Stage 1. In order to "sync up" the working instance and the proof we (1) introduce $\ell + \overline{b}_C \geq 1$ to the proof with the witness $\{b_C \rightarrow 0\}$, (2) update O^1 by adding $w^C \ell - w^C b_C$, (3) remove the constraint $\ell + \overline{b}_C \geq 1$ with the witness $\{b_C \rightarrow 0\}$, and (4) remove the constraint $\overline{\ell} + b_C \geq 1$ with witness $\{b_C \rightarrow 1\}$. The same steps are logged for all soft unit clauses of $\mathcal{F}_1^W$ obtained during Stage 2. In the following stages, the preprocessor will operate on an objective-centric MaxSAT instance whose clauses correspond exactly to the core constraints of the proof.

Stage 4: Preprocessing on the Objective-Centric Representation.

During preprocessing on the objective-centric representation, more simplification techniques are applied to the working objective-centric instance and logged to the proof. We implemented proof logging for a wide range of preprocessing techniques. These include MaxSAT versions of rules commonly used in SAT solving like bounded variable elimination (BVE) [EB05, SP04], bounded variable addition [MHB13], blocked clause elimination [JBH10], subsumption elimination, self-subsuming resolution [EB05, OGMS02], failed literal elimination [Fre95, LB01, ZM88], and equivalent literal substitution [Bra04, Li00, VG05]. We also cover MaxSAT-specific preprocessing rules like TrimMaxSAT [PRB21], (group)-subsumed literal (or label) elimination (SLE) [BSJ16, KBSJ17], intrinsic at-most-ones [IMM19, IBJ22], binary core removal (BCR) [Gim64, KBSJ17], label matching [KBSJ17], and hardening [ABGL12, IBJ22, MHM12]. Here we give examples for BVE, SLE, label matching, and BCR—the rest are detailed in Appendix A. In the following descriptions, let (F, O) be the current objective-centric working instance.

Bounded Variable Elimination (BVE) [EB05, SP04]. BVE eliminates from F a variable x that does not appear in the objective by replacing all clauses in which either x or $\overline{x}$ appears with the non-tautological clauses in $\{C \vee D \mid C \vee x \in F, D \vee \overline{x} \in F\}$.

An application of BVE is logged as follows: (1) each non-tautological constraint $\mathrm{PB}(C \vee D)$ is added by summing the existing constraints $\mathrm{PB}(C \vee x)$ and $\mathrm{PB}(D \vee \overline{x})$ and saturating, after which (2) each constraint of the form $\mathrm{PB}(C \vee x)$ and $\mathrm{PB}(D \vee \overline{x})$ is deleted with the witness $x \to 1$ or $x \to 0$, respectively.

Label Matching [KBSJ17]. Label matching allows merging pairs of objective variables that can be deduced to not both be set to 1 by optimal solutions. Assume that (i) F contains the clauses $C \vee b_C$ and $D \vee b_D$, (ii) b_C and b_D are objective variables with the same coefficient w in O, and (iii) $C \vee D$ is a tautology. Then label matching replaces b_C and b_D with a fresh variable b_{CD}, i.e., replaces $C \vee b_C$ and $D \vee b_D$ with $C \vee b_{CD}$ and $D \vee b_{CD}$ and adds $-wb_C - wb_D + wb_{CD}$ to O.

As $C \vee D$ is a tautology there is some literal ℓ such that $\overline{\ell} \in C$ and $\ell \in D$. Label matching is logged via the following steps: (1) introduce the constraint $\overline{b}_C + \overline{b}_D \geq 1$ with the witness $\{b_C \to \ell, b_D \to \overline{\ell}\}$, (2) introduce the constraints $b_{CD} + \overline{b}_C + \overline{b}_D \geq 2$ and $\overline{b}_{CD} + b_C + b_D \geq 1$ by redundance; these correspond to $b_{CD} = b_C + b_D$ which holds even though the variables are binary due to the constraint added in the first step, (3) update the objective by adding $-wb_C - wb_D + wb_{CD}$ to it, (4) introduce the constraints $\mathrm{PB}(C \vee b_{CD})$ and $\mathrm{PB}(D \vee b_{CD})$ which are RUP, (5) delete $\mathrm{PB}(C \vee b_C)$ and $\mathrm{PB}(D \vee b_D)$ with the witness $\{b_C \to \overline{\ell}, b_D \to \ell\}$, (6) delete the constraint $b_{CD} + \overline{b}_C + \overline{b}_D \geq 2$ with the witness $\{b_C \to 0, b_D \to 0\}$ and $\overline{b}_{CD} + b_C + b_D \geq 1$ with the witness $\{b_C \to 1, b_D \to 0\}$, (7) delete $\overline{b}_C + \overline{b}_D \geq 1$ with the witness $\{b_C \to 0\}$.

Subsumed Literal Elimination (SLE) [BSJ16, IBJ22]. Given two non-objective variables x and y such that (i) $\{C \mid C \in F, y \in C\} \subseteq \{C \mid C \in F, x \in C\}$ and (ii) $\{C \mid C \in F, \overline{x} \in C\} \subseteq \{C \mid C \in F, \overline{y} \in C\}$, subsumed literal elimination (SLE) allows fixing $x = 1$ and $y = 0$. This is proven by (1) introducing $x \geq 1$ and $\overline{y} \geq 1$, both with witness $\{x \rightarrow 1, y \rightarrow 0\}$, (2) simplifying the constraint database via propagation, and (3) deleting the constraints introduced in the first step as neither x nor y appears in any other constraints after simplification.

If x and y are objective variables, the application of SLE additionally requires that: (iii) the coefficient in the objective of x is at most as high as the coefficient of y. Then the value of x is not fixed as it would incur cost. Instead, only $y = 0$ is fixed and y removed from the objective. Intuitively, conditions (i) and (ii) establish that the values of x and y can always be flipped to 0 and 1, respectively, without falsifying any clauses. If neither of the variables is in the objective, this flip does not increase the cost of any solutions. Otherwise, condition (iii) ensures that the flip does not make the solution worse, i.e., increase its cost.

Binary Core Removal (BCR) [Gim64, KBSJ17]. Assume that the following four prerequisites hold: (i) F contains a clause $b_C \lor b_D$ for two objective variables b_C and b_D, (ii) b_C and b_D have the same coefficient w in O, (iii) the negations b_C and b_D do not appear in any clause of F, and (iv) both b_C and b_D appear in at least one other clause of F but not together in any other clause of F. Binary core removal replaces all clauses containing b_C or b_D with the non-tautological clauses in $\{C \lor D \lor b_{CD} \mid C \lor b_C \in F, D \lor b_D \in F\}$, where b_{CD} is a fresh variable, and modifies the objective function by adding $-wb_C - wb_D + wb_{CD} + w$ to it.

BCR is logged as a combination of the so-called *intrinsic at-most-ones* technique [IMM19, IBJ22] and BVE. Applying intrinsic at most ones on the variables b_C and b_D introduces a new clause $(\overline{b}_C \lor \overline{b}_D \lor b_{CD})$ and adds $-wb_C - wb_D + wb_{CD} + w$ to the objective. Our proof for intrinsic at most ones is the same as the one presented in [BBN+23]. As this step removes b_C and b_D from the objective, both can now be eliminated via BVE.

Stage 5: Constant Removal and Output.

After objective-centric preprocessing, the final objective-centric instance (F^3, O^3) is converted back to a WCNF instance. Before doing so, the constant term W_3 of O^3 is removed by introducing a fresh variable b^{W_3}, and setting $F^4 = F^3 \land (b^{W_3})$ and $O^4 = O^3 - W_3 + W_3 b^{W_3}$. This step is straightforward to prove.

Finally, the preprocessor outputs the WCNF instance $\mathcal{F}_P^W = (F^4, F_S^P)$ that has F^4 as hard clauses. the set F_S^P of soft clauses consists of a unit soft clause $(\overline{\ell})$ of weight c for each term $c \cdot \ell$ in O^4. The preprocessor also outputs the final proof of the fact that the minimum-cost of solutions to the pseudo-Boolean formula PB(F^0) under O^0 is the same as that of PB(F^4) under O^4, i.e. that $\textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}^W)) = \textsc{opt}(\textsc{ObjMaxSAT}(\mathcal{F}_P^W))$.

3.2 Worked Example of Certified Preprocessing

Table 1: *Example proof produced by a certifying preprocessor. The column (ID) refers to constraint IDs in the pseudo-Boolean proof. The column (Step) indexes all proof logging steps and is used when referring to the steps in the discussion. The letter ω is used for the witness substitution in redundance-based strengthening steps.*

Step	ID	Type	Justification	Objective
1	(1)	add $x_1 + x_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
2	(2)	add $\overline{x}_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
3	(3)	add $x_3 + \overline{x}_4 + b_1 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
4	(4)	add $x_4 + \overline{x}_5 + b_2 \geq 1$	input	$x_1 + 2b_1 + 3b_2$
	Unit propagation: fix $x_2 = 0$, constraint (2)			
5	(5)	add $x_1 \geq 1$	(1) + (2)	$x_1 + 2b_1 + 3b_2$
6		delete (1)	RUP	$x_1 + 2b_1 + 3b_2$
7		delete (2)	$\omega\colon \{x_2 \rightarrow 0\}$	$x_1 + 2b_1 + 3b_2$
	Unit propagation; fix $x_1 = 1$, constraint (5)			
8		add $-x_1 + 1$ to obj.	(5)	$2b_1 + 3b_2 + 1$
9		delete (5)	$\omega\colon \{x_1 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
	BVE: eliminate x_4			
10	(6)	add $x_3 + b_1 + \overline{x}_5 + b_2 \geq 1$	(3) + (4)	$2b_1 + 3b_2 + 1$
11		delete (3)	$\omega\colon \{x_4 \rightarrow 0\}$	$2b_1 + 3b_2 + 1$
12		delete (4)	$\omega\colon \{x_4 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
	Subsumed literal elimination: $\overline{b}_2$			
13	(7)	add $\overline{b}_2 \geq 1$	$\omega\colon \{b_2 \rightarrow 0, b_1 \rightarrow 1\}$	$2b_1 + 3b_2 + 1$
14		add $-3b_2$ to obj.	(7)	$2b_1 + 1$
15	(8)	add $x_3 + b_1 + \overline{x}_5 \geq 1$	(6) + (7)	$2b_1 + 1$
16		delete (6)	RUP	$2b_1 + 1$
17		delete (7)	$\omega\colon \{b_2 \rightarrow 0\}$	$2b_1 + 1$
	Remove objective constant			
18	(9)	add $b_3 \geq 1$	$\omega\colon \{b_3 \rightarrow 1\}$	$2b_1 + 1$
19		add $b_3 - 1$ to obj.	(9)	$2b_1 + b_3$

We give a worked-out example of certified preprocessing of the instance $\mathcal{F}^W = (F_H, F_S)$ where $F_H = \{(x_1 \vee x_2), (\overline{x}_2)\}$ and three soft clauses: $(\overline{x}_1)$ with weight 1, $(x_3 \vee \overline{x}_4)$ with weight 2, and $(x_4 \vee \overline{x}_5)$ with weight 3. The proof for one possible execution of the preprocessor on this input instance is detailed in Table 1.

During Stage 1 (Steps 1–4 in Table 1), the core constraints of the proof are initialized to contain the four constraints corresponding to the hard and non-unit soft clauses of $\mathcal{F}^W$ (IDs (1)–(4) in Table 1), and the objective to $x_1 + 2b_1 + 3b_2$, where b_1 and b_2 are fresh variables added to the non-unit soft clauses of $\mathcal{F}^W$.

During Stage 2 (Steps 5–9), the preprocessor fixes $x_2 = 0$ via unit propagation by removing x_2 from the clause $(x_1 \vee x_2)$, and then removing the unit clause $(\overline{x}_2)$.

The justification for fixing $x_2 = 0$ are Steps 5–7. Next the preprocessor fixes $x_1 = 1$ which (i) removes the hard clause (x_1), and (ii) increases the lower bound on the optimal cost by 1. The justification for fixing $x_1 = 1$ are Steps 8 and 9 of Table 1. At this point—at the end of Stage 2—the working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ has $F_H^1 = \{\}$ and $F_S^1 = \{(x_3 \vee \overline{x}_4), (x_4 \vee \overline{x}_5)\}$.

In Stage 3, the preprocessor converts its working instance into the objective-centric representation (F, O) where $F = \{(x_3 \vee \overline{x}_4 \vee b_1), (x_4 \vee \overline{x}_5 \vee b_2)\}$ and $O = 2b_1 + 3b_2 + 1$, which exactly matches the core constraints and objective of the proof after Step 9. Thus, in this instance, the conversion does not result in any proof logging steps. Afterwards, during Stage 4 (Steps 10–17), the preprocessor applies BVE in order to eliminate x_4 (Steps 10–12) and SLE to fix b_2 to 0 (Steps 13–17). Finally, Steps 18 and 19 represent Stage 5, i.e., the removal of the constant 1 from the objective. After these steps, the preprocessor outputs the preprocessed instance $\mathcal{F}_P^W = (F_H^P, F_S^P)$, where $F_H^P = \{(x_3 \vee \overline{x}_5 \vee b_1), (b_3)\}$ and F_S^P contains two clauses: $(\overline{b}_1)$ with weight 2, and $(\overline{b}_3)$ with weight 1.

4 Verified Proof Checking for Preprocessing Proofs

This section presents our new workflow for formally verified, end-to-end proof checking of MaxSAT preprocessing proofs based on pseudo-Boolean reasoning; an overview of this workflow is shown in Figure 2. To realize this workflow, we extended the VeriPB tool and its proof format to support a new *output section* for declaring (and checking) reformulation guarantees between input and output PBO instances (Section 4.1); we similarly modified CakePB [GMM+24] a verified proof checker to support the updated proof format (Section 4.2); finally, we built a verified frontend, CakePBwcnf, which mediates between MaxSAT WCNF instances and PBO instances (Section 4.3). Our formalization is carried out in the HOL4 proof assistant [SN08] using CakeML tools [GMKN17, MO14, TMK+19] to obtain a verified executable implementation of CakePBwcnf.

In the workflow in Figure 2, the MaxSAT preprocessor produces a reformulated output WCNF together with a proof of equioptimality with the input WCNF. This proof is elaborated by VeriPB and then checked by CakePBwcnf, resulting in a verified *verdict*—in case of success, the input and output WCNFs are equioptimal. This workflow also supports verified checking of WCNF MaxSAT solving proofs (where the output parts of the flow are omitted).

4.1 Output Section for Pseudo-Boolean Proofs

Given an input PBO instance (F, O), the VeriPB proof system as described in Section 2.1 maintains the invariant that the core constraints C (and current objective) are equioptimal to the input instance. Utilizing this invariant, the new *output section* for VeriPB proofs allows users to optionally specify an output PBO instance (F', O') at the end of a proof. This output instance is claimed to be a reformulation

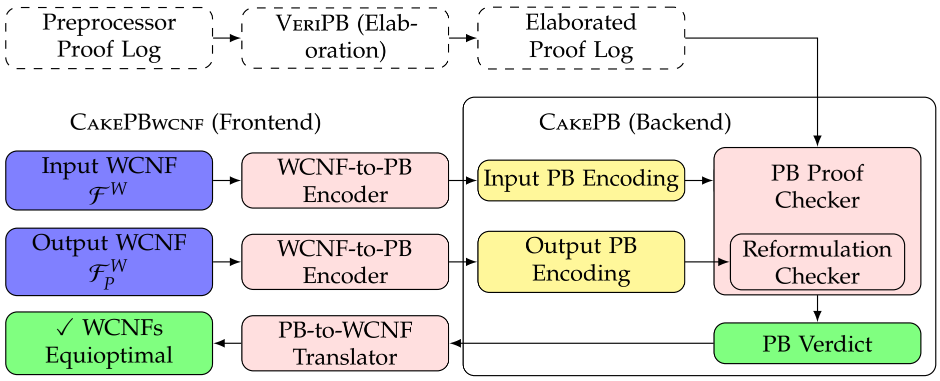


Figure 2: *Workflow for end-to-end verified MaxSAT preprocessing proof checking.*

of the input which is either: (i) *derivable*, i.e., satisfiability of F implies satisfiability of F', (ii) *equisatisfiable* to F, or (iii) *equioptimal* to (F, O). These are increasingly stronger claims about the relationship between the input and output instances. After checking a pseudo-Boolean derivation, VeriPB runs reformulation checking which, e.g., for equioptimality, checks that $C \subseteq F'$, $F' \subseteq C$, and that the respective objective functions are syntactically equal after normalization; other reformulation guarantees are checked analogously.

The VeriPB tool supports an *elaboration* mode [GMM+24], where in addition to checking the proof it also converts it from *augmented format* to *kernel format*. The augmented format contains syntactic sugar rules to facilitate proof logging for solvers and preprocessors like MaxPre, while the kernel format is supported by the formally verified proof checker CakePB. The new output section is passed unchanged from augmented to kernel format during elaboration.

4.2 Verified Proof Checking for Reformulations

There are two main verification tasks involved in extending CakePB with support for the output section. The first task is to verify soundness of all cases of reformulation checking. Formally, the equioptimality of an input PBO instance *fml*, *obj* and its output counterpart *fml′*, *obj′* is specified as follows:

$$\begin{aligned}&\mathsf{sem\_output}\ \mathit{fml}\ \mathit{obj}\ \mathsf{None}\ \mathit{fml}'\ \mathit{obj}'\ \mathsf{Equioptimal}\ \stackrel{\text{def}}{=}\\ &\forall v.\ (\exists w.\ \mathsf{satisfies}\ w\ \mathit{fml} \wedge \mathsf{eval\_obj}\ \mathit{obj}\ w \leq v) \iff\\ &\quad (\exists w'.\ \mathsf{satisfies}\ w'\ \mathit{fml}' \wedge \mathsf{eval\_obj}\ \mathit{obj}'\ w' \leq v)\end{aligned}$$

This definition says that, for all values v, the input instance has a satisfying assignment with objective value less than or equal to v iff the output instance also has such an assignment; note that this implies (as a special case) that *fml* is satisfiable iff *fml′* is satisfiable. The verified correctness theorem for CakePB says that *if* CakePB successfully checks a pseudo-Boolean proof in kernel format and prints a verdict declaring equioptimality, then the input and output instances are indeed equioptimal as defined in `sem_output`.

The second task is to develop verified optimizations to speedup proof steps which occur frequently in preprocessing proofs; some code hotspots were also identified by profiling the proof checker against proofs generated by MaxPre. Similar (unverified) versions of these optimizations are also used in VeriPB. These optimizations turned out to be necessary in practice—they mostly target steps which, when naïvely implemented, have quadratic (or worse) time complexity in the size of the constraint database.

Optimizing Reformulation Checking. The most expensive step in reformulation checking for the output section is to ensure that the core constraints $\mathcal{C}$ are included in the output formula and vice versa (possibly with permutations and duplicity). Here, CakePB normalizes all pseudo-Boolean constraints involved to a canonical form and then copies both $\mathcal{C}$ and the output formula into respective array-backed hash tables for fast membership tests.

Optimizing Redundance and Checked Deletion Rules. A naïve implementation of these two rules would require iterating over the entire constraints database when checking all subproofs in (1) for the right-hand-side constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\upharpoonright_\omega \cup \{O \geq O\upharpoonright_\omega\}$. An important observation here is that preprocessing proofs frequently use substitutions ω that only involve a small number of variables (often a single variable, which in addition is fresh in the important special case of *reification* constraints $z \Leftrightarrow C$ encoding that z is true precisely when the constraint C is satisfied). Consequently, most of the constraints $(\mathcal{C} \cup \mathcal{D} \cup \{C\})\upharpoonright_\omega$ can be skipped when checking redundance because they are unchanged by the substitution. Similarly, the constraint $O \geq O\upharpoonright_\omega$ is expensive to construct when the objective O contains many terms, but this construction can be skipped if no variables being substituted occur in O. CakePB stores a lazily-updated mapping of variables to their occurrences in the constraint database and the objective, which it uses to detect these cases.

The occurrence mapping just discussed is crucial for performance due to the frequency of steps involving witnesses for preprocessing proofs, but incurs some memory overhead in the checker. More precisely, every variable occurrence in any constraint in the database corresponds to exactly one ID in the mapping. Thus, the overhead of storing the mapping is in the worst case quadratic in the number of constraints, but it is still linear in the total space usage for the constraints database.

4.3 Verified WCNF Frontend

The CakePBwcnf frontend mediates between MaxSAT WCNF problems and pseudo-Boolean optimization problems native to CakePB. Accordingly, the correctness of CakePBwcnf is stated in terms of MaxSAT semantics, i.e., the encoding, underlying pseudo-Boolean semantics, and proof system are all formally verified.

$$
\begin{aligned}
&\mathsf{sat\_hard}\ w\ \mathit{wfml} \overset{\text{def}}{=} \forall C.\ \mathsf{mem}\ (0,C)\ \mathit{wfml} \Rightarrow \mathsf{sat\_clause}\ w\ C\\
&\mathsf{weight\_clause}\ w\ (n,C) \overset{\text{def}}{=} \text{if } \mathsf{sat\_clause}\ w\ C \text{ then } 0 \text{ else } n\\
&\mathsf{cost}\ w\ \mathit{wfml} \overset{\text{def}}{=} \mathsf{sum}\ (\mathsf{map}\ (\mathsf{weight\_clause}\ w)\ \mathit{wfml})\\
&\mathsf{opt\_cost}\ \mathit{wfml} \overset{\text{def}}{=} \text{if } \neg\exists w.\ \mathsf{sat\_hard}\ w\ \mathit{wfml} \text{ then None}\\
&\qquad\qquad\qquad\quad \text{else Some } (\mathsf{min}_{\mathsf{set}}\ \{\ \mathsf{cost}\ w\ \mathit{wfml} \mid \mathsf{sat\_hard}\ w\ \mathit{wfml}\ \})
\end{aligned}
$$

Figure 3: *Formalized semantics for MaxSAT WCNF problems.*

$$
\begin{array}{ll}
\vdash \mathsf{wfml\_to\_pbf}\ \mathit{wfml} = (\mathit{obj},\mathit{pbf}) \land & \vdash \mathsf{wfml\_to\_pbf}\ \mathit{wfml} = (\mathit{obj},\mathit{pbf}) \land\\
\quad \mathsf{satisfies}\ w\ (\mathsf{set}\ \mathit{pbf}) \Rightarrow & \quad \mathsf{sat\_hard}\ w\ \mathit{wfml} \Rightarrow\\
\qquad \exists w'.\ \mathsf{sat\_hard}\ w'\ \mathit{wfml} \land & \qquad \exists w'.\ \mathsf{satisfies}\ w'\ (\mathsf{set}\ \mathit{pbf}) \land\\
\qquad\qquad \mathsf{cost}\ w'\ \mathit{wfml} \le \mathsf{eval\_obj}\ \mathit{obj}\ w & \qquad\qquad \mathsf{eval\_obj}\ \mathit{obj}\ w' = \mathsf{cost}\ w\ \mathit{wfml}
\end{array}
$$

$$
\begin{aligned}
&\vdash \mathsf{full\_encode}\ \mathit{wfml} = (\mathit{obj},\mathit{pbf}) \land \mathsf{full\_encode}\ \mathit{wfml}' = (\mathit{obj}',\mathit{pbf}') \land\\
&\quad \mathsf{sem\_output}\ (\mathsf{set}\ \mathit{pbf})\ \mathit{obj}\ \mathsf{None}\ (\mathsf{set}\ \mathit{pbf}')\ \mathit{obj}'\ \mathsf{Equioptimal} \Rightarrow\\
&\qquad \mathsf{opt\_cost}\ \mathit{wfml} = \mathsf{opt\_cost}\ \mathit{wfml}'
\end{aligned}
$$

Figure 4: *Correctness theorems for the WCNF-to-PB encoding.*

In order to trust CakePBwcnf, one *only* has to carefully inspect the formal definition of MaxSAT semantics shown in Figure 3 to make sure that it matches the informal definition in Section 2.2. Here, each clause C is paired with a natural number n, where $n = 0$ indicates a hard clause and when $n > 0$ it is the weight of C. The optimal cost of a weighted CNF formula *wfml* is None (representing ∞) if no satisfying assignment to the hard clauses exist; otherwise, it is the minimum cost among all satisfying assignments to the hard clauses.

There and Back Again. CakePBwcnf contains a verified WCNF-to-PB encoder implementing the encoding described in Section 2.2. Its correctness theorems are shown in Figure 4, where the two lemmas in the top row relate the satisfiability and cost of the WCNF to its PB optimization counterpart after running wcnf_to_pbf (and vice versa), see Observation 1. Using these lemmas, the final theorem (bottom row) shows that equioptimality for two (encoded) PB optimization problems can be *translated* back to equioptimality for the input and preprocessed WCNFs.

Putting Everything Together. The final verification step is to specialize the end-to-end machine code correctness theorem for CakePB to the new frontend. The resulting theorem for CakePBwcnf is shown abridged in Figure 5; a detailed explanation of similar CakeML-based theorems is available elsewhere [GMM+24, THM23] so we do not go into details here. Briefly, the theorem says that whenever the verdict string "s VERIFIED OUTPUT EQUIOPTIMAL" is printed (as a suffix) to

```
⊢ cake_pb_wcnf_run cl fs mc ms ⇒
    ∃ out err.
      extract_fs fs (cake_pb_wcnf_io_events cl fs) =
        Some (add_stdout (add_stderr fs err) out) ∧
      (length cl = 4 ∧ isSuffix "s VERIFIED OUTPUT EQUIOPTIMAL\n" out ⇒
        ∃ wfml wfml′.
          get_fml fs (el 1 cl) = Some wfml ∧ get_fml fs (el 3 cl) = Some wfml′ ∧
          opt_cost wfml = opt_cost wfml′)
```

Figure 5: *Abridged final correctness theorem for CakePBwcnf.*

the standard output by an execution of CakePBwcnf, then the two input files given on the command line parsed to equioptimal MaxSAT WCNF instances.

5 Experiments

We upgraded the MaxSAT preprocessor MaxPre 2.1 [IBJ22, JBIJ23, KBSJ17] to MaxPre 2.2, which produces proof logs in the VeriPB format [BMM$^+$23]. MaxPre 2.2 is available at the MaxPre 2 repository [Maxa]. The generated proofs were elaborated using VeriPB [Ver] and then checked by the verified proof checker CakePBwcnf. As benchmarks we used the 558 weighted and 572 unweighted MaxSAT instances from the MaxSAT Evaluation 2023 [Max23].

The experiments were conducted on 11th Gen Intel(R) Core(TM) i5-1145G7 @ 2.60GHz CPUs with 16 GB of memory, a solid state drive as storage, and Rocky Linux 8.5 as operating system. Each benchmark ran exclusively on a node and the memory was limited to 14 GB. The time for MaxPre was limited to 300 seconds. There is an option to let MaxPre know about this time limit, but we did not use this option since MaxPre then decides which techniques to try based on how much time remains. This would have made it very hard to get reliable measurements of the overhead when proof logging is switched on in the preprocessor. The time limits for both VeriPB and CakePBwcnf were set to 6000 seconds to get as many instances checked as possible.

The main focus of our evaluation was the default setting of MaxPre, which does not use some of the techniques mentioned in Section 3 (or Appendix A). We also conducted experiments with all techniques enabled to check the correctness of the proof logging implementation for all preprocessing techniques. The data and source code from our experiments can be found in [IOT$^+$24].

The goal of the experiments was to answer the following questions:

RQ1. How much extra time is required to write the proof for the preprocessor?

RQ2. How long does proof checking take compared to proof generation?

To answer the first question, in Figure 6 we compare MaxPre with and without proof logging. In total, 1081 instances were successfully preprocessed by

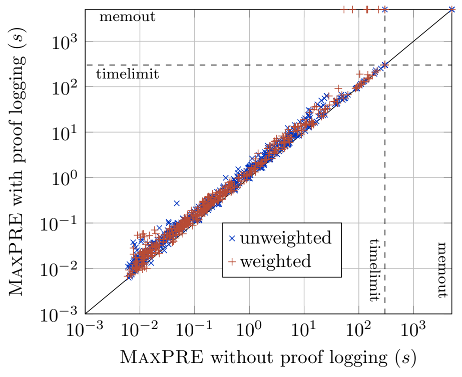


Figure 6: *Proof logging overhead for MaxPre.*

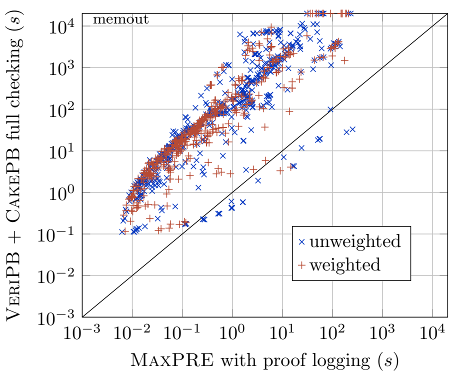


Figure 7: *MaxPre vs. combined proof checking running time.*

MaxPre without proof logging. With proof logging enabled, 8 fewer instances were preprocessed due to either time- or memory-outs. For the successfully preprocessed instances, the geometric mean of the proof logging overhead is 46% of the running time, and 95% of the instances were preprocessed with proof logging in at most twice the time required without proof logging.

Our comparison between proof generation and proof checking is based on the 1073 instances for which preprocessing with proof logging was successful. Out of these, 1021 instances were successfully checked and elaborated by VeriPB. For 991 instances the verdicts were confirmed by the formally verified proof checker CakePBwcnf, with the remaining instances being time- or memory-outs. This shows the practical viability of our approach, as the vast majority of preprocessing proofs were checked within the resource limits.

A scatter plot comparing the running time of MaxPre with proof logging enabled against the combined checking process is shown in Figure 7. For the combined checking time, we only consider the instances that have been successfully checked by CakePBwcnf. In the geometric mean, the time for the combined verified checking pipeline of VeriPB elaboration followed by CakePBwcnf checking is 113× the preprocessing time of MaxPre. A general reason for this overhead is that the preprocessor has more MaxSAT application-specific context than the pseudo-Boolean checker, so the preprocessor can log proof steps without performing the actual reasoning while the checker must ensure that those steps are sound in an application-agnostic way. An example for this is reification: as the preprocessor knows its reification variables are fresh, it can easily emit redundance steps that witness on those variables; but the checker has to verify freshness against its own database. Similar behaviour has been observed in other applications of pseudo-Boolean proof logging [GMNO22, HOGN24].

To analyse further the causes of proof checking overhead, we also compared VeriPB to CakePBwcnf. The checking of the elaborated kernel proof with CakePBwcnf is 6.7× faster than checking and elaborating the augmented proof with VeriPB. This suggests that the bottleneck for proof checking is VeriPB; VeriPB *without* elaboration is about 5.3× slower than CakePBwcnf. As elaboration is a necessary step before running CakePBwcnf, improving the performance of VeriPB would benefit the performance of the pipeline as a whole. One specific feature that seems desirable would be to augment RUP rule applications with LRAT-style hints [CHH$^+$17], so that VeriPB would not need to perform unit propagation to elaborate RUP steps to cutting planes derivations. Though these types of engineering challenges are important to address, they are beyond the scope of the current paper and we have to leave them as future work.

6 Conclusion

In this work, we show how to use pseudo-Boolean proof logging to certify correctness of the MaxSAT preprocessing phase, extending previous work for the main solving phase in unweighted model-improving solvers [VDB22] and general core-guided solvers [BBN$^+$23]. As a further strengthening of previous work, we present a fully formally verified toolchain which provides end-to-end verification of correctness.

In contrast to SAT solving, there is a rich variety of techniques in maximum satisfiability solving, and it still remains to design pseudo-Boolean proof logging methods for general, weighted, model-improving MaxSAT solvers [ES06, LP10, PRB18] and *implicit hitting set (IHS)* MaxSAT solvers [DB11, DB13] with *abstract cores* [BBP20]. Nevertheless, our work adds further weight to the conclusion that pseudo-Boolean reasoning seems like a very promising foundation for MaxSAT proof logging. We are optimistic that this work is another step on the path towards general adoption of proof logging in the context of SAT-based optimization.

Acknowledgements

This work has been financially supported by the University of Helsinki Doctoral Programme in Computer Science DoCS, the Research Council of Finland under grants 342145 and 346056, the Swedish Research Council grants 2016-00782 and 2021-05165, the Independent Research Fund Denmark grant 9040-00389B, the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and by A*STAR, Singapore. Part of this work was carried out while some of the authors participated in the extended reunion of the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2023 at the Simons Institute for the Theory of Computing at UC Berkeley. We also acknowledge useful discussions at the Dagstuhl workshops 22411 *Theory and Practice of SAT and Combinatorial Solving* and 23261 *SAT Encodings and Beyond*.

The computational experiments were enabled by resources provided by LUNARC at Lund University.

Appendix A Complete Overview of Proof Logging for MaxSAT Preprocessing

In this appendix, we provide a complete overview of proof logging for the preprocessing techniques implemented by MaxPre. As we already presented proof logging for bounded variable elimination, subsumed literal elimination, label matching and binary core removal in Section 3 of the paper, we do not present those techniques here. In addition, we do not include intrinsic at-most-ones (even though implemented in MaxPre), as it is already discussed in [BBN+23].

A.1 Fixing Variables

Many of the preprocessing techniques can fix variables (or literals) to either 0 or 1. We describe here the generic procedure that is invoked when a variable is fixed. Assume that a preprocessing technique decides to fix $\ell = 1$ for a literal ℓ. Then, in the preprocessor, each clause $C \vee \overline{\ell}$ is replaced by clause C, i.e., falsified literal $\overline{\ell}$ is removed. Additionally, each clause $C \vee \ell$ is removed (as they are satisfied when $\ell = 1$).

In the proof, we do the following. First, the technique that fixes $\ell = 1$, ensures that constraint $\ell \geq 1$ is in the core constraints of the proof. It may be that $\ell \geq 1$ is already in the core constraints of the proof (i.e. instance has a unit clause (ℓ)), or it may be that $\ell \geq 1$ needs to be introduced as a new constraint. The details on how $\ell \geq 1$ is introduced depends on the specific technique that is fixing $\ell = 1$. Now, assuming $\ell \geq 1$ is in the core constraints, the following procedure is invoked.

(1) If ℓ or $\overline{\ell}$ appears in the objective function, the objective function is updated.

(2) For each clause $C \vee \overline{\ell}$, the constraint $\mathrm{PB}(C)$ is introduced as a sum of $\mathrm{PB}(C \vee \overline{\ell})$ and $\ell \geq 1$.

(3) Each constraint $\mathrm{PB}(C \vee \ell)$ is deleted (as a RUP constraint).

(4) Finally, the core constraint $\ell \geq 1$ is deleted last with witness $\{\ell \to 1\}$.

A.2 Preprocessing on the Initial WCNF Representation

We explain the preprocessing techniques that can be applied during preprocessing on the WCNF representation, detailing especially how the different types of clauses are handled. The preprocessing techniques applied on the WCNF representation only modify a clause C by either removing a literal ℓ from C or removing C entirely. With this intuition, given an input WCNF instance $\mathcal{F}^W = (F_H, F_S)$ and a working instance $\mathcal{F}_1^W = (F_H^1, F_S^1)$ each clause in $\mathcal{F}_1^W$ is one of the following three types:

(1) A hard clause $C \in F_H^1$ that is a subset or equal to a hard clause $C \subseteq C^{\text{orig}} \in F_H$ of $\mathcal{F}^W$.

(2) An *originally unit soft clause*, i.e., a soft clause $C \in F_S^1$ that is equal to a unit soft clause in F_S.

(3) An *originally non-unit soft clause*, i.e., $C \in F_S^1$ that is a subset or equal to a non-unit soft clause $C \subseteq C^{\text{orig}} \in F_S$ of $\mathcal{F}^W$.

With this we next detail how the preprocessing rules permitted on the WCNF representation are logged. In the following, we assume a fixed working WCNF instance.

Duplicate Clause Removal.

In the paper we discussed how to log the removal of two duplicate clauses C and D when: (i) both are hard, or (ii) C is hard and D is an originally non-unit soft clause. Here we detail the remaining cases.

Assume first that both C and D are originally non-unit duplicate soft clauses with weights w^C and w^D, respectively. Then the proof has the core constraints $\text{PB}(C \vee b_C)$ and $\text{PB}(D \vee b_D)$ and its objective the terms $w^C b_C$ and $w^D b_D$. The removal of D is logged as follows.

(1) Introduce the constraints: $\overline{b}_C + b_D \geq 1$ with the witness $\{b_C \to 0\}$ and $b_C + \overline{b}_D \geq 1$, with the witness $\{b_D \to 0\}$ to the core set. These encode $b_C = b_D$.

(2) Update the objective by adding $-w^C b_C + w^C b_D$ to it, conceptually increasing the coefficient of b_D by w^C.

(3) Remove the constraints introduced in step (1) using the same witnesses.

(4) Remove the (RUP) constraint $\text{PB}(D \vee b_C)$.

If $C = (\overline{\ell})$ is originally a unit soft clause but $D = (\overline{\ell})$ is originally a non-unit soft clause, then the core constraints of the proof include constraint $\text{PB}(\ell \vee b_D)$ and the objective of the proof the terms $w^C \ell$ and $w^D b_D$. The removal of D is logged similarly to the previous case with the literal b_C replaced with ℓ.

The case of two duplicate originally unit soft clauses does not require proof logging since the corresponding terms in the objective are automatically summed.

Tautology Removal.

If a clause is a tautology, it is also a RUP clause. Thus, a tautological hard clause is simply deleted. The removal of a tautological soft clause additionally requires updating the objective.

More specifically, assume C is a tautological soft clause of weight w^C. Then C is originally non-unit, so the proof has a constraint $\text{PB}(C \vee b_C)$ and its objective the term $w^C b_C$. The removal of C is logged with the following steps:

(1) Delete the (RUP) constraint PB($C \lor b_C$).

(2) Introduce the constraint $\overline{b}_C \geq 1$ with witness $\{b_C \to 0\}$ and move the new constraint to the core set.

(3) Update the objective by adding $-w^C b_C$ to it.

(4) Remove the constraint introduced in step (2) with the same witness.

Unit Propagation of Hard Clauses.

If the instance contains a (hard) unit clause (l), the literal l is fixed to 1 with the method of fixing variables described in Section A.1.

Removal of Empty Soft Clauses.

If the instance contains an empty soft clause C—either as input or as a consequence of e.g., unit propagation—it is removed and the lower bound increased by its weight w^C. If C was originally non-unit, the core constraints of the proof contain the constraint $b_C \geq 1$ and the objective the term $w^C b_C$. The removal of C is logged by the following steps:

(1) Update the objective by adding $-w^C b_C + w^C$.

(2) Delete the constraint $b_C \geq 1$ with the witness $\{b_C \to 1\}$.

If $C = (\ell)$ is an originally unit soft clause the objective is updated in conjunction with the literal ℓ getting fixed to 0, as described in Section A.1. Thus, no further steps are required.

Blocked Clause Elimination (BCE) [JBH10].

Our implementation of BCE considers a clause $C \lor \ell$ blocked (on the literal ℓ) if for each clause $D \lor \overline{\ell}$ there is a literal $\ell' \in D$ for which $\overline{\ell'} \in C$.

When preprocessing on the objective-centric representation, BCE considers only literals ℓ for which neither ℓ nor $\overline{\ell}$ appears in the objective function. During initial WCNF preprocessing stage, there are no requirements for literal ℓ. (Notice that whenever there is a unit clause $(\overline{\ell})$, $C \lor \ell$ is not blocked on the literal ℓ.)

The removal of a blocked clause is logged as the deletion of the corresponding constraint PB($C \lor \ell$) with the witness $\{\ell \to 1\}$. If $C \lor \ell$ is an (originally non-unit) soft clause, the objective function is also updated exactly as with tautology removal.

Subsumption Elimination.

A clause D is subsumed by the clause C if $C \subseteq D$. Whenever the subsuming clause C is hard, D is removed as a RUP clause. If D is soft, the objective function is updated exactly as with tautology removal.

A.3 Preprocessing on Objective-Centric Representation

We detail how the preprocessing techniques that are applied on the objective-centric representation (F, O) of the working instance are logged. In addition to these, the preprocessor can also apply the techniques detailed in Section A.2.

TrimMaxSAT [PRB21].

The TrimMaxSAT technique heuristically looks for a set of literals N s.t. every solution ρ to F assigns each $\ell \in N$ to 0, or more formally, F entails the unit clause $(\overline{\ell})$. All such literals are fixed by the generic procedure (recall Section A.1). The literals to be fixed are identified by iterative calls to an (incremental) SAT solver [ES03, MLM21] under different assumptions.

In order to log the TrimMaxSAT technique we log the proof produced by each SAT solver call into the derived set of constraints in our PB proof. After the set N is identified, we make $|N|$ extra SAT calls, one for each $\ell \in N$. Each call is made assuming the value of ℓ to 1. Due to the properties of TrimMaxSAT and SAT-solvers, the result will be UNSAT, after which $\overline{\ell} \geq 1$ will be RUP w.r.t to the current set of core and derived constraints. As such it is added and moved to core in order to invoke the generic variable fixing procedure. Finally, when TrimMaxSAT will not be used any more, all constraints added to the derived set by the SAT solver are removed.

Self-Subsuming Resolution (SSR) [EB05, OGMS02].

Given clauses $C \vee l$ and $D \vee \overline{\ell}$ such that C subsumes D and ℓ is not in the objective, SSR substitutes D for $D \vee \overline{\ell}$. The proof has two steps: (1) Introduce PB(D) as a new RUP constraint. (2) Remove PB($D \vee \overline{\ell}$) as it is RUP.

Group-Subsumed Label Elimination (GSLE) [KBSJ17].

Let b be an objective variable that has the coefficient c^b in O, and L a set of objective variables such that each $b_i \in L$ has coefficient c^i in O. Assume then that (i) $c^b \geq \sum_{b_i \in L} c^i$, (ii) the negation of b or any variables in L do not appear in any clauses, and (iii) $\{C \mid b \in C\} \subseteq \{D \mid \exists b \in L : b \in D\}$. Then, an application of GSLE fixes $b = 0$. To prove an application of GSLE, we introduce the constraint $\overline{b} \geq 1$ with the witness $\{b \rightarrow 0, b_i \rightarrow 1 \mid b_i \in L\}$, and invoke the generic variable fixing procedure detailed in Section A.1 to fix $b = 0$.

Bounded Variable Addition (BVA) [MHB13].

Consider a set of literals M_{lit} and a set of clauses $M_{cls} \subseteq F$, such that for all $\ell \in M_{lit}$ and $C \in M_{cls}$, each clause $(C \setminus M_{lit} \cup \{\ell\})$ is either in F or a tautology. Then an application of BVA adds the clauses $S_x = \{(\ell \vee x) \mid l \in M_{lit}\}$ and $S_{\overline{x}} = \{(C \setminus M_{lit}) \cup \{\overline{x}\} \mid C \in M_{cls}\}$, and removes the clauses $C \setminus M_{lit}$.

An application of BVA is logged as follows: (1) Add the constraint PB(C) for each $C \in S_{\overline{x}}$ with the witness $\{x \rightarrow 0\}$. (2) Add the constraint PB(C) for each $C \in S_x$ with the witness $\{x \rightarrow 1\}$. (3) Delete each constraint PB(C) for $C \in M_{cls}$ as a RUP constraint.

Structure-based Labelling [KBSJ17].

Given an objective variable b and a clause C that is blocked on the literal ℓ, when $b = 1$, an application of structure-based labelling replaces C with $C \vee b$. The proof is logged as follows: (1) Introduce the constraint PB($C \vee b$) that is RUP. (2) Delete the constraint PB(C) with the witness $\{\ell \rightarrow 1\}$.

Failed Literal Elimination (FLE) [Fre95, LB01, ZM88].

A literal ℓ is failed (denoted $\ell \vdash_{\text{UP}} \bot$) if setting $\ell = 1$ allows unit propagation to derive a conflict (i.e., an empty clause). An application of FLE fixes $\ell = 0$ when ℓ is a failed literal for which $\overline{\ell}$ is not in the objective.

In addition to standard FLE, MaxPre implements an extension that also fixes a literal $\ell = 0$ if: (i) $\overline{\ell}$ is not in the objective function (ii) each clause in F that contains ℓ also contains some other literal ℓ' that is implied by ℓ by unit propagation (denoted $\ell \vdash_{\text{UP}} \ell'$), i.e., setting $\ell = 1$ also fixes $\ell' = 1$ after a sequence of unit propagation steps is applied.

Logging FLE. For a failed literal ℓ the constraint $\overline{\ell} \geq 1$ is RUP. For the extended technique the constraint $\overline{\ell} \geq 1$ is introduced with the witness $\{\ell \rightarrow 0\}$. Afterwards the generic procedure for fixing literals described in Section A.1 is invoked.

Implied Literal Detection.

If both a literal ℓ_1 and its negation $\overline{\ell}_1$ imply another literal ℓ_2 by unit propagation (i.e., propagating either $\ell = 1$ or $\ell = 0$ also propagates $\ell_2 = 1$), the preprocessor fixes $\ell_2 = 1$.

As an extension to this technique, the preprocessor also fixes $\ell_2 = 1$ if (i) ℓ_1 implies ℓ_2 by unit propagation, (ii) neither ℓ_1 nor ℓ_2 appear in the objective function in either polarity, and (iii) each clause containing $\overline{\ell}_2$ also contains some other literal ℓ' that is implied by $\overline{\ell}_1$ by unit propagation.

Logging Implied Literals. For some intuition, note that $\ell_1 \vdash_{\text{UP}} \ell_2$ does not in general imply $\overline{\ell}_2 \vdash_{\text{UP}} \overline{\ell}_1$. Thus, there is no guarantee that $\ell_2 \geq 1$ would be RUP. Given that $\ell_1 \vdash_{\text{UP}} \ell_2$ and $\overline{\ell}_1 \vdash_{\text{UP}} \ell_2$, the proof is instead logged as follows:

(1) Add $\overline{\ell}_1 + \ell_2 \geq 1$ and $\ell_1 + \ell_2 \geq 1$ that are both RUP.

(2) Introduce the constraint $\ell_2 \geq 1$ by divide the sum of constraints introduced in step (1) by 2. Move the new constraint to the core constraints.

(3) Delete the constraints introduced in step (1).

(4) Invoke the generic procedure detailed in Section A.1 to fix $\ell_2 = 1$.

The extended technique is logged by first adding the constraint $\ell_1 + \ell_2 \geq 1$ with the witness $\{\ell_2 \to 1\}$. For some intuition, if the constraint is falsified, the assumptions guarantee that $\ell' = 1$ so the value of ℓ_2 can be flipped without falsifying other constraints.

Equivalent Literal Substitution [Bra04, Li00, VG05].

If $\ell_1 \vdash_{\text{UP}} \ell_2$ and $\overline{\ell}_1 \vdash_{\text{UP}} \overline{\ell}_2$, the equivalent literal technique substitutes ℓ_1 with ℓ_2. As an extension to this technique, the same substitution is applied also in cases where the following three conditions hold: (i) $\ell_1 \vdash_{\text{UP}} \ell_2$, (ii) neither ℓ_1 nor ℓ_2 appear in the objective function in either polarity, and (iii) $\overline{\ell}_1$ implies some other literal in each clause containing $\overline{\ell}_2$ by unit propagation.

Logging Equivalent Literals. An application of equivalent literal substitution is logged as follows.

(1) Introduce the clauses $\overline{\ell}_1 + \ell_2 \geq 1$ and $\ell_1 + \overline{\ell}_2 \geq 1$ as RUP. In the case of the extended technique, $\ell_1 + \overline{\ell}_2 \geq 1$ is added with the witness $\{\ell_2 \to 0\}$.

(2) For each clause $C \vee \ell_1$, replace $\text{PB}(C \vee \ell_1)$ with $\text{PB}(C \vee \ell_2)$ with the RUP rule.

(3) For each clause $C \vee \overline{\ell}_1$, replace $\text{PB}(C \vee \overline{\ell}_1)$ with $\text{PB}(C \vee \overline{\ell}_2)$ with the RUP rule.

(4) If ℓ_1 or $\overline{\ell}_1$ appear in the objective function, replace them with ℓ_2 and $\overline{\ell}_2$, respectively.

(5) Remove the constraints introduced in step (1).

Hardening [ABGL12, IBJ22, MHM12].

Given an upper bound UB for the optimal cost of (F, O) and an objective variable b that has a coefficient $w^b > UB$ in O, hardening fixes $b = 0$. Proof logging for hardening has been previously studied in [BBN+23]. In [BBN+23], however, the hardening is done with the presence of so-called objective-improving constraints, i.e., constraints of form $O \leq UB - 1$, where UB is the cost of the best currently known solution. In the context of preprocessing where the preprocessor should provide an equioptimal instance as an output, introducing objective-improving constraints to the instance is not possible. Instead, given a solution ρ to F with cost $O(\rho) = UB$ and an objective variable b with $w^b > UB$, we introduce the constraint $\overline{b} \geq 1$ with ρ as the witness and then invoke the generic procedure for fixing variables, as detailed in Section A.1.

A.4 Conversion to WCNF — Renaming Variables

In the final stage of preprocessing, MaxPre converts the instance to WCNF. The conversion removes the objective constant as described in Section 3.1 of the main paper. Additionally, the conversion 'renames' (some of) the variables.

There are two reasons for renaming variables. The first is to remove any gaps in the indexing of variables. In WCNF, variables are named with integers. During preprocessing, some variables in the instance might have been eliminated from the instance. At the end MaxPre compacts the range of variables to be continuous and start from 1. The second reason for renaming variables is to sync names between WCNF and the pseudo-Boolean proof. In the pseudo-Boolean proofs, the naming scheme of variables is different, valid variable names include, for instance, `x1, x2, y15, _b4`. When a WCNF instance is converted to a pseudo-Boolean instance, the variable `i` of the WCNF instance is mapped to the variable `xi` of the pseudo-Boolean instance. For jth non-unit soft clause of a WCNF instance, the conversion introduces a variable `_bj`. During preprocessing, the 'proof logger' of MaxPre takes care of mapping MaxPre variables to correct variable names in proof. In the end, however, MaxPre produces an output WCNF file, and at this point, each variable `i` of WCNF instance should again correspond to variable `xi` of proof. Thus, for example, all `_b`-variables are replaced with `x`-variables.

Logging variable naming. Assume that the instance has a set of variables V and for each $x \in V$, we wish to use name $f(x)$ instead of x in the end. We do proof logging for variable renaming in two phases. (1) For each $x \in V$, introduce temporary variable t_x, set $x = t_x$ and then 'move' all the constraints and the objective function to the temporary namespace. The original constraints and encodings for $x = t_x$ are then removed. (2) For each $x \in V$, introduce $f(x) = t_x$, and 'move' the constraints and the objective to the final namespace. The temporary constraints and encodings are then removed.

A.5 On Solution Reconstruction and Instances Solved During Preprocessing

Finally, we note that while the focus of this work has been on certifying the preservation of the costs of solutions, in practice our certified preprocessor also allows reconstructing a minimum-cost solution to the input. More precisely, consider an input WCNF instance $\mathcal{F}^W$, a preprocessed instance $\mathcal{F}_P^W$, and an optimal solution ρ_p to $\mathcal{F}_P^W$. Then MaxPre can compute an optimal solution ρ to $\mathcal{F}^W$ in linear time with respect to the number of preprocessing steps performed. More details can be found in [KBSJ17].

Importantly, the optimality of a reconstructed solution can be easily verified without considering how the reconstruction is implemented in practice; given that we have verified the equioptimality of $\mathcal{F}^W$ and $\mathcal{F}_P^W$, and that ρ_p is an optimal solution to $\mathcal{F}_P^W$, the optimality of reconstructed ρ to $\mathcal{F}^W$ can be verified by checking that (i) ρ indeed is a solution to $\mathcal{F}^W$ (ii) The cost of ρ w.r.t. $\mathcal{F}^W$ is equivalent to the cost of ρ_p w.r.t. $\mathcal{F}_P^W$.

On a related note, MaxPre can actually solve some instances during preprocessing, either by: (i) determining that the hard clauses do not have solutions, or (ii) computing an optimal solution to some working instance. In practice (i) happens by the derivation of the unsatisfiable empty (hard) clause and (ii) by the removal of every single clause from the working instance. We have designed the preprocessor to always terminate with an output WCNF and a proof of equioptimality rather than producing different kinds of proofs.

If an empty hard clause is derived, the preprocessing is immediately terminated and an output WCNF instance containing a single hard empty clause produced. Additionally, an empty constraint $0 \geq 1$ is added to the proof and all other core constraints deleted by the RUP rule. Notice how the proof of equioptimality between the input and output can in this case be seen as a proof of infeasibility of the input hard clauses.

If all clauses are removed from the working instance, MaxPRE terminates and outputs the instance obtained after constant removal (recall Stage 5 in Section 3) on an instance without other clauses.

References

[ABGL12] Carlos Ansótegui, María Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-based weighted MaxSAT solvers. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 86–101. Springer, October 2012.

[ABM+11] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, Christine Rizkallah, and Pascal Schweitzer. An introduction to certifying algorithms. *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(6):287–293, December 2011.

[BBN+23] 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.

[BBP20] Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSat solving. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, July 2020.

[BGMN23] 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*.

[BHvMW21] 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*. IOS Press, 2nd edition, February 2021.

[Bie06] Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006.

[BJ19] Jeremias Berg and Matti Järvisalo. Unifying reasoning and core-guided search for maximum satisfiability. In *Proceedings of the 16th European Conference on Logics in Artificial Intelligence (JELIA '19)*, volume 11468 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2019.

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT '10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, July 2010.

[BLM07] Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for Max-SAT. *Artificial Intelligence*, 171(8-9):606–618, 2007.

[BMM$^+$23] Bart Bogaerts, Ciaran McCreesh, Magnus O. Myreen, Jakob Nordström, Andy Oertel, and Yong Kiam Tan. Documentation of VeriPB and CakePB for the SAT competition 2023. Available at `https://satcompetition.github.io/2023/checkers.html`, March 2023.

[BN21] Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [BHvMW21], chapter 7, pages 233–350.

[Bra04] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(1):52–59, 2004.

[BSJ16] Jeremias Berg, Paul Saikko, and Matti Järvisalo. Subsumed label elimination for maximum satisfiability. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI '16)*, volume 285 of *FAIA*, pages 630–638. IOS Press, 2016.

[CCT87] William Cook, Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, November 1987.

[CHH$^+$17] 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.

[CMS17] 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.

[DB11] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP '11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, September 2011.

[DB13] Jessica Davies and Fahiem Bacchus. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT '13)*, volume 7962 of *Lecture Notes in Computer Science*, pages 166–181. Springer, July 2013.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT '05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, June 2005.

[EGMN20] 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.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. In Ofer Strichman and Armin Biere, editors, *First International Workshop on Bounded Model Checking, (BMC '03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 543–560. Elsevier, 2003.

[ES06] Niklas Eén and Niklas Sörensson. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, March 2006.

[FMSV20] Yuval Filmus, Meena Mahajan, Gaurav Sood, and Marc Vinyals. MaxSAT resolution and subcube sums. In *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT '20)*, volume 12178 of *Lecture Notes in Computer Science*, pages 295–311. Springer, July 2020.

[Fre95] Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[Gim64] James F. Gimpel. A reduction technique for prime implicant tables. In *Proceedings of the 5th Annual Symposium on Switching Circuit Theory and Logical Design, (SWCT '64)*, pages 183–191. IEEE Computer Society, 1964.

[GMKN17] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *Proceedings of the 26th European Symposium on Programming (ESOP '17)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, April 2017.

[GMM$^+$20] 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.

[GMM$^+$24] 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 368h AAAI Conference on Artificial Intelligence (AAAI '24)*, pages 8038–8047, February 2024.

[GMN20] 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.

[GMN22] 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.

[GMNO22] 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.

[GN03] 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.

[GN21] 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.

[Goc22] 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`.

[HHW13a] 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.

[HHW13b] 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.

[HOGN24] Alexander Hoen, Andy Oertel, Ambros Gleixner, and Jakob Nordström. Certifying MIP-based presolve reductions for 0–1 integer linear programs. In *Proceedings of the 21st International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '24)*, May 2024. To appear.

[IBJ22] Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Proceedings of the 11th International Joint Conference on Automated Reasoning (IJCAR '22)*, volume 13385 of *Lecture Notes in Computer Science*, pages 75–94. Springer, August 2022.

[IMM19] Alexey Ignatiev, António Morgado, and João Marques-Silva. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.

[IOT$^+$24] Hannes Ihalainen, Andy Oertel, Yong Kiam Tan, Jeremias Berg, Matti Järvisalo, Magnus O. Myreen, and Jakob Nordström. Experimental Repository for "Certified MaxSAT Preprocessing", February 2024.

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010.

[JBIJ23] Christoph Jabs, Jeremias Berg, Hannes Ihalainen, and Matti Järvisalo. Preprocessing in SAT-based multi-objective combinatorial optimization. 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 18:1–18:20, 2023.

[KBSJ17] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. MaxPre: An extended MaxSAT preprocessor. In *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT '17)*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017.

[LB01] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[Li00] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 291–296. AAAI Press / The MIT Press, 2000.

[LNOR11] Javier Larrosa, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A framework for certified Boolean branch-and-bound optimization. *Journal of Automated Reasoning*, 46(1):81–102, 2011.

[LP10] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, July 2010.

[Maxa] MaxPre 2 : MaxSAT preprocessor. `https://bitbucket.org/coreo-group/maxpre2`.

[Maxb] MaxSAT evaluations: Evaluating the state of the art in maximum satisfiability solver technology. `https://maxsat-evaluations.github.io/`.

[Max23] MaxSAT evaluation 2023. `https://maxsat-evaluations.github.io/2023`, July 2023.

[MHB13] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In *8th International Haifa Verification Conference (HVC '12), Revised Selected Papers*, volume 7857 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2013.

[MHM12] António Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for MaxSAT. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT '12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2012.

[MIB$^+$19] António Morgado, Alexey Ignatiev, María Luisa Bonet, João P. Marques-Silva, and Samuel R. Buss. DRMaxSAT with MaxHS: First contact. 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 239–249. Springer, July 2019.

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

[MM11] António Morgado and João Marques-Silva. On validating Boolean optimizers. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence, (ICTAI '11)*, pages 924–926, 2011.

[MM23] 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.

[MMN24] 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)*, May 2024. To appear.

[MMNS11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, May 2011.

[MO14] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2–3):284–315, January 2014.

[OGMS02] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP '02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[PCH20] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. 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)*, pages 137–144, November 2020.

[PCH21] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. A proof builder for Max-SAT. In *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing (SAT '21)*, volume 12831 of *Lecture Notes in Computer Science*, pages 488–498. Springer, July 2021.

[PCH22] Matthieu Py, Mohamed Sami Cherif, and Djamal Habet. Proofs and certificates for Max-SAT. *Journal of Artificial Intelligence Research*, 75:1373–1400, December 2022.

[PRB18] 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.

[PRB21] Tobias Paxian, Pascal Raiola, and Bernd Becker. On preprocessing for weighted MaxSAT. In *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI '21)*, volume 12597 of *Lecture Notes in Computer Science*, pages 556–577. Springer, 2021.

[SAT] The International SAT Competitions web page. `http://www.satcompetition.org`.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs '08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, August 2008.

[SP04] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT '04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. Verified propagation redundancy and compositional UNSAT checking in CakeML. *International Journal on Software Tools for Technology Transfer*, 25:167–184, February 2023. Preliminary version in *TACAS '21*.

[TMK$^+$19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29:e2:1–e2:57, February 2019.

[VDB22] 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.

[Ver] VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/software/VeriPB`.

[VG05] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Annals of Mathematics and Artificial Intelligence*, 43(1):239–253, 2005.

[WHH14] 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.

[ZM88] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI '88)*, pages 155–160. AAAI Press / The MIT Press, 1988.