LUND UNIVERSITY

Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning

Gocht, Stephan

2022

Certifying Correctness for Combinatorial Algorithms

by Using Pseudo-Boolean Reasoning

by Stephan Gocht

LUND
UNIVERSITY

Thesis for the degree of Doctor of Philosophy in Engineering
Thesis advisors: Prof. Jakob Nordström
Faculty opponent: Prof. Marijn J. H. Heule

To be presented, with the permission of the Faculty of Engineering of Lund University, for public criticism in E:A at the Department of Computer Science (Ole Römers Väg 3 in Lund) on Friday, the 10th of June 2022 at 14:15.

Organization
LUND UNIVERSITY
Department of Computer Science
Box 118
SE–221 00 LUND
Sweden

Document name
DOCTORAL DISSERTATION

Date of disputation
2022-06-10

Sponsoring organization

Author(s)
Stephan Gocht

Title and subtitle
Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning

Abstract
Over the last decades, dramatic improvements in combinatorial optimisation algorithms have significantly impacted artificial intelligence, operations research, and other areas. These advances, however, are achieved through highly sophisticated algorithms that are difficult to verify and prone to implementation errors that can cause incorrect results. A promising approach to detect wrong results is to use certifying algorithms that produce not only the desired output but also a certificate or proof of correctness of the output. An external tool can then verify the proof to determine that the given answer is valid. In the Boolean satisfiability (SAT) community, this concept is well established in the form of proof logging, which has become the standard solution for generating trustworthy outputs. The problem is that there are still some SAT solving techniques for which proof logging is challenging and not yet used in practice. Additionally, there are many formalisms more expressive than SAT, such as constraint programming, various graph problems and maximum satisfiability (MaxSAT), for which efficient proof logging is out of reach for state-of-the-art techniques.

This work develops a new proof system building on the cutting planes proof system and operating on pseudo-Boolean constraints (0-1 linear inequalities). We explain how such machine-verifiable proofs can be created for various problems, including parity reasoning, symmetry and dominance breaking, constraint programming, subgraph isomorphism and maximum common subgraph problems, and pseudo-Boolean problems. We implement and evaluate the resulting algorithms and a verifier for the proof format, demonstrating that the approach is practical for a wide range of problems. We are optimistic that the proposed proof system is suitable for designing certifying variants of algorithms in pseudo-Boolean optimisation, MaxSAT and beyond.

Key words
certifying algorithms, 0-1 linear inequalities, combinatorial algorithms, proof logging

Classification system and/or index terms (if any)

Supplementary bibliographical information

Language
English

ISSN and key title
1404-1219

ISBN
978-91-8039-267-9 (print)
978-91-8039-268-6 (pdf)

Recipient's notes

Number of pages
228

Price
0 SEK

Security classification

Certifying Correctness for Combinatorial Algorithms

by Using Pseudo-Boolean Reasoning

by Stephan Gocht

LUND
UNIVERSITY

A doctoral thesis at a university in Sweden takes either the form of a single, cohesive research study (monograph) or a summary of research papers (compilation thesis), which the doctoral student has written alone or together with one or several other author(s).

In the latter case the thesis consists of two parts. An introductory text puts the research work into context and summarizes the main points of the papers. Then, the research publications themselves are reproduced, together with a description of the individual contributions of the authors. The research papers may either have been already published or are manuscripts at various stages (in press, submitted, or in draft).

Cover illustration front: A problem is solved by an algorithm that does not only produce an answer (41) but also a certificate. Then a verifier uses the certificate to verify the answer to the problem. However, the used algorithm contains a failure, also known as a bug, and the verification fails.

Funding information: Stephan Gocht was supported by the Swedish Research Council grant 2016-00782. Part of this work was carried out while taking part in the semester program Satisfiability: Theory, Practice, and Beyond in the spring of 2021 at the Simons Institute for the Theory of Computing at UC Berkeley.

Faculty of Engineering, Department of Computer Science

ISBN: 978-91-8039-267-9 (print)
ISBN: 978-91-8039-268-6 (pdf)
ISSN: 1404-1219
Dissertation 69, 2022
LU-CS-DISS: 2022-03

Printed in Sweden by Tryckeriet i E-huset, Lund University, Lund 2022

Contents

Acknowledgements

I thank my family and friends for supporting me throughout my educational journey. I thank Jo Devriendt and Jan Elffers for the exciting discussions that helped improve the performance of VeriPB, and Bart Bogaerts and Ciaran McCreesh for their significant contributions to the proof format and for providing interesting applications for it. I thank Jonas Conneryd, Leo Krull, Andy Oertel and Susanna F. de Rezende for helping with proof-reading the thesis. Any errors that remain are my sole responsibility. Last but not least, I thank my PhD supervisor Jakob Nordström without whom this work would not have been possible.

Popular Science Summary

Do you trust your computer? Computers may not act as intended, even without assuming malicious intent, due to faulty software or hardware. As automated systems make more decisions in our world, the question of trust becomes even more relevant. If some decision or insight is essential in the non-digital world, then we do not trust the decision blindly. Instead, we require proof, be it a receipt in accounting for later audit, repeated counting of ballots in an election, or a scientist who needs to write down his ideas and convince colleagues during peer-review. Wouldn't it be nice if we could verify the computations of a computer in a similar way? Similar to how a school teacher would verify the answer to an exam exercise like the following.

"Due to a global pandemic, two small countries vaccinate their population within three months. Together, both countries have 12 million inhabitants, and the first country vaccinates three times as many people per month as the second country. Can you provide the average number of people vaccinated per month in each country?"

Now the student should start modelling the problem using variables a for the average number of vaccinations per month in the first country and b for the second country. And then he should extract the following equations from the exercise

$$3 \cdot b = a \tag{1}$$
$$3 \cdot (a + b) = 12. \tag{2}$$

While the well-behaved student starts solving the problem, the lazy student has already finished: He just wrote "Yes, I can provide the number." as an answer. This answer is technically correct, the question only asked if the student *can* provide an answer. However, it will be very unsatisfactory for the teacher, who was expecting a concrete solution to verify the student's claim. Given the mathematical formulation of the exercise above, a solution is $a = 3, b = 1$. Now the teacher can verify the "yes" answer by checking that both equations are satisfied with this assignment without having to compute the solution himself. (And the teacher does not need to worry that the solution is copied due to the distance requirements during a pandemic.)

Although this is an elementary example, it allows us to introduce the fundamental concepts underlying this thesis, as shown in Figure 1. We will

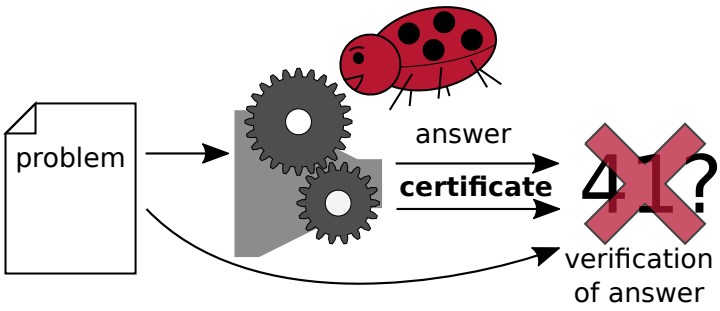


Figure 1: A problem is solved by an algorithm that does not only produce an answer (41) but also a certificate. Then a verifier uses the certificate to verify the answer to the problem. However, the used algorithm contains a failure, also known as a bug, and the verification fails.

start with a formal, mathematical formulation of a problem. Instead of a student, we use a computer program to solve the problem, referred to as a solver. The solver will produce an answer (in the case of a decision problem as above, the answer is "yes" or "no") as well as a proof or certificate that this answer is correct (in our example, the certificate is $a = 3, b = 1$). Instead of a teacher, we will have a second computer program that verifies the answer to the problem using the certificate.

Producing and checking the proof for the previous exercise was quite simple, but how about the following extension: "Both countries together receive at most 3 million doses of the vaccine per month. Can they still vaccinate at least 12 million people within three months?" Using variables a and b as before, the problem is modelled as

$$3 \cdot (a + b) \geq 12 \tag{3}$$

$$3 \geq (a + b). \tag{4}$$

Providing a proof for the answer of this problem is more challenging as the answer to this exercise is "no" because there is no solution to these inequalities. Instead of using a solution as certificate, the certificate should be a sequence of rules or steps that we need to agree are correct and that are easy to check. For example, if $3 \cdot (a + b) \geq 12$ holds, then so does

$$(a + b) \geq 4, \tag{5}$$

which we get by dividing both sides of the inequality by 3. It is almost obvious that there is no solution because $a + b$ cannot be smaller than 3 and also greater than 4. To make this even more clear, we can add the inequalities (4) and (5) to obtain $3 + (a + b) \geq 4 + (a + b)$, which can be simplified to $0 \geq 1$. We have an obvious contradiction. The rules we used are (a) dividing by a positive number, (b) adding constraints and (c) simplifying constraints. It can be shown that each rule guarantees to preserve solutions, which means that if there was a solution to the set of inequalities before applying one of these rules, then there is a solution afterwards. However, the constraint $0 \geq 1$ is never true, which means that the original set of constraints must have been inconsistent. The proof that there is no solution could look something like "divide (3) by 3 and add (4)". This proof will derive $0 \geq 1$ as discussed above, assuming constraints are simplified implicitly.

This thesis describes how we can obtain such certificates for various computer programs and will study the following questions: How can we model problems solved by different computer programs? What are the easily verifiable rules that we want to use? How can we prove the correctness of the reasoning of various programs with these rules? How efficient is it to generate and to verify these proofs?

List of Publications and Author Contributions

This section contains an overview of the included papers and the author contributions. The main content of the thesis starts with Section 1.

This thesis is based on the following publications, referred to by their letters. Note that author ordering is alphabetic except for Paper C and hence does not reflect author contribution, which is discussed later in this section.

A Stephan Gocht and Jakob Nordström. "Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '21)*. Vol. 35. 5. 2021, pp. 3768–3777.

B Bart Bogaerts, Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Certified Symmetry and Dominance Breaking for Combinatorial Optimisation". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '22)*. Vol. 36. 2022. To appear.

C 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 AAAI Conference on Artificial Intelligence (AAAI '21)*. Vol. 35. 5. 2021, pp. 3768–3777.

D Stephan Gocht, Jakob Nordström, Ruben Martins and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". In: *Proceedings of the 25nd International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*. 2022. To appear.

E Jan Elffers, Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Justifying All Differences Using Pseudo-Boolean Reasoning". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '20)*. Vol. 34. 02. AAAI Press, 2020, pp. 1486–1494.

F 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)*. 2022. To appear.

G Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solu-

tions". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, (IJCAI '20)*. 2020, pp. 1134–1140.

H 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)*. Vol. 12333. Lecture Notes in Computer Science. Springer, 2020, pp. 338–357.

I Stephan Gocht, Jakob Nordström and Amir Yehudayoff. "On Division Versus Saturation in Pseudo-Boolean Solving". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, (IJCAI '19)*. 2019, pp. 1711–1718.

I also published the following papers during my PhD, that are not included in the thesis.

- Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht and Jakob Nordström. "In between resolution and cutting planes: a study of proof systems for pseudo-Boolean SAT solving". In: *Proceedings of the 25nd International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*. Springer. 2018, pp. 292–310.
- Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström and Laurent Simon. "Seeking Practical CDCL Insights from Theoretical SAT Benchmarks." In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, (IJCAI '18)*. 2018, pp. 1300–1308.
- Mate Soos, Stephan Gocht and Kuldeep S. Meel. "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling". In: *Proceedings of the International Conference on Computer Aided Verification (CAV '20)*. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 463–484.

A visual representation of the author's contributions can be found in the following table. For all papers, there always was an ongoing discussion between all authors at every stage but especially while defining the concept(s).

Paper	Concept	Implementation		Evaluation	Writing
		Verifier	Solver		
Paper A	●	●	●	●	◕
Paper B	◑	●	○	◑	◑
Paper C	◑		◔	◔	◑
Paper D	◕	●	◑	◑	◕
Paper E	◑	●	○	○	◔
Paper F	◔	●	○	○	○
Paper G	◑	●	○	○	◔
Paper H	◔	●	○	○	○
Paper I	◑			●	◕

● Lead and did almost all the work

◕ Lead and did a majority of the work

◑ Contributed to a majority of the work

◔ Contributed to a minority of the work

○ Did not contribute to the work, except for proof reading

Concept Coming up with the ideas of the paper

Implementation Implementing the software described in the paper

Evaluation Conducting the evaluation described in the paper

Writing Drafting and editing the paper

A detailed discussion of contributions follows.

Paper A

Stephan Gocht was the lead author on this paper and did the bulk of the work while discussing the progress with Jakob Nordström. Stephan Gocht wrote most of the paper, and Jakob Nordström provided feedback on the writing. Jakob Nordström wrote the introduction.

Paper B

After Stephan Gocht presented the redundancy-based strengthening rule and explained why it is problematic in connection with optimisation, Bart Bogaerts suggested transferring the technique used in redundancy-based strengthening to the concept of dominating assignments from constraint programming, which led to the dominance rule as well as adjustments to the redundancy rule to support optimisation fully. Stephan Gocht discovered that deletion can be a problem in this proof system and how to fix the proof system. The correctness proof for the proof system was a collaborative effort. Stephan Gocht developed the concrete format for the proof system, as well as the implementation in *VeriPB*. How to do proof logging for the different applications was a collaborative effort.

Paper C

The idea of expressing core-guided MaxSAT techniques came up in a discussion between Stephan Gocht, Emir Demirović and Jakob Nordström, after which Jo Devriendt visited Emir Demirović and Peter Stuckey to learn more about the techniques in-depth and to create a knowledge transfer. Jo Devriendt implemented the technique while having regular in-depth discussions about algorithm design and debugging with Stephan Gocht and Jakob Nordström. The evaluation was performed by Jo Devriendt, with some detailed experiments on a subset of benchmarks by Stephan Gocht. The latter does not appear in the paper but influenced the conclusions. Stephan Gocht wrote the first version of the preliminaries and contributions section, which was later heavily edited and improved by all authors.

Paper D

Stephan Gocht developed proof logging for sequential counter and (generalised) totaliser encoding, with regular discussions with Jakob Nordström. Stephan Gocht also developed the framework to systematise proof logging for the different encodings. Andy Oertel took care of proof logging for the binary adder. For the other encodings, Stephan Gocht provided prototype implementations for proof logging. Andy Oertel did the final implementation of the generalised totaliser and binary adder network. Ruben Martins provided knowledge about different encodings, implemented proof logging for the sequential counter and performed the evaluation. All authors dis-

cussed the benchmark set and data to measure. Stephan Gocht provided a more detailed empirical analysis of asymptotic behaviour. Stephan Gocht had the lead in writing the theoretical part of the paper, except for the binary adder encoding, written mainly by Andy Oertel. There was a detailed discussion about the structure of the paper with all authors.

Paper E

The authors collectively realised that proof logging via cutting planes could express the reasoning of various solvers, such as a solver for subgraph isomorphism. Ciaran McCreesh suggested focusing first on the all-different constraint. Stephan Gocht developed the proof format for expressing cutting planes proofs and implemented a verifier. Ciaran McCreesh implemented the solver, but backtracking was harder to implement than expected, as the solver would not have all data readily available. As a result of a discussion, Jakob Nordström and Stephan Gocht decided to provide a rule for reverse unit propagation to simplify proof logging. After this was implemented in the verifier by Stephan Gocht, Ciaran McCreesh could quickly finish the implementation of the solver, which confirmed that having such a rule is indeed helpful. Ciaran McCreesh performed the experiments and also wrote most of the paper with feedback from the other authors.

Paper F

Stephan Gocht and Jakob Nordström assisted with problems that arose for proof logging while using different encodings for integer variables simultaneously and discussed encoding choices with Ciaran McCreesh. Apart from this, most of the work on this paper was performed by Ciaran McCreesh. The paper used the previously developed verifier and profited from ongoing improvements of the verifier by Stephan Gocht.

Paper G

Figuring out how to do proof logging for subgraph isomorphism was a highly interactive effort between all authors. The solver was implemented and evaluated by Ciaran McCreesh. Stephan Gocht improved the verifier to handle the large proof files constructed. Ciaran McCreesh wrote the paper with feedback from the other authors.

Paper H

Stephan Gocht assisted with how to encode connectedness. He also provided general tips, such as how to construct an at-most one constraint from its clausal representation to obtain a clique lower bound from a colouring. Apart from this, most of the work on this paper was performed by the other authors. The paper used the previously developed verifier, which was extended to deal with optimisation and profited from ongoing improvements of the verifier by Stephan Gocht.

Paper I

Stephan Gocht developed the proof for the equivalence between saturation-based cutting planes with and without the limitation of only using generalized resolution and the separation of saturation-based cutting planes from division-based cutting planes. Jakob Nordström and Amir Yehudayoff developed the lower bound on the number of divisions required to replace a single saturation. Stephan Gocht later improved the parameters of this result. The instances for the evaluation were designed and evaluated by Stephan Gocht. Stephan Gocht wrote the paper, except for the introduction and the conclusion. All authors helped to improve the writing.

Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning

1 Introduction

Generally speaking, users of any software have to trust that the software is producing the correct answer. However, designing correct software is difficult and costly, as is demonstrated repeatedly through critical failures (also known as bugs). At the same time, we can see dramatic improvements in the amount and difficulty of tasks that modern algorithms can solve. As a result, such algorithms get deployed in high-stakes environments where bugs can lead to large financial losses, like in combinatorial auctions [LMS17], or even where lives are at stake, for example, when finding an optimal allocation for kidney transplants in kidney exchange programs [MO12].

A classical approach for detecting software bugs is to use extensive testing. However, software testing can only detect bugs but never show their absence. Another approach is formally verifying that the software adheres to a given specification for all possible inputs. While there has been tremendous progress in formal verification, it requires expert knowledge and tends to be time-consuming. Therefore, formal verification for complex systems seems out of reach. Instead, we will focus on a middle ground between testing and formal verification with the concept of *certifying algorithms* [McC+11]. A certifying algorithm does not only produce an answer to a given problem but also a machine-verifiable certificate of correctness. An external verifier can then verify this certificate independently of the original algorithm. Importantly, the verifier is usually a much simpler software than the original program, making it easier to implement it correctly or even to formally verify it.

Using a certifying algorithm can be beneficial throughout the software life cycle. The verification can ease test case creation during development because the correct answer does not need to be known. Additionally, it can detect bugs even if the provided answer was correct but was obtained using incorrect reasoning, for example, by ignoring corner cases. And finally, if there is a bug, then the first incorrect step in the certificate can help locate the origin of the bug. Once the software is deployed, checking the produced certificates can increase the trust in the produced results and can detect hardware failures. Additionally, the certificates can potentially be used to audit an answer at a later point in time or to analyse and improve the software.

Certifying algorithms are well established for determining the satisfiabil-

ity of propositional formulas (SAT) in the form of *proof logging*. There are numerous proof logging formats for SAT solving such as *RUP* [GN03], *TraceCheck* [Bie06], and *DRAT* [HHW13a; HHW13b; WHH14]. Some formats such as *LRAT* [Cru+17] and *GRIT* [CMS17] have formally verified checkers. *DRAT* is the de facto standard used in the main track of the annual SAT competition, which introduced mandatory proof logging because some submitted solvers provided the correct answer, but were quicker on some instances due to bugs. A novel use case of proof logging is to use the produced proof as an analysis tool, for example, to understand the power and benefits of different heuristics in SAT solvers [Elf+18] or to use systematic machine learning to replace handcrafted heuristics [SKM19].

Although *DRAT* is a very powerful proof system, there are still techniques in SAT solving for which it is not known if efficient proof logging is feasible with *DRAT*. For example, reasoning with parities (equalities modulo 2) using Gaussian elimination [SNC09; HJ12], or breaking symmetries [Dev+16]. The former is crucial for solving cryptographic problems, approximate model counting and almost-uniform sampling. The latter is crucially used in the winning solver of the 2016 SAT competition's no-limits track, where no proof logging is required. While there are attempts for *DRAT* proof logging of parity reasoning [PR16] and symmetry breaking [HHW15], the proposed proof logging is complex to implement and comes with a considerable polynomial overhead during solving with the result that no solver has adopted them.

When considering other combinatorial problems such as constraint programming, clique, subgraph isomorphism, pseudo-Boolean optimisation or mixed-integer programming, certification becomes even more challenging as solvers for these problems assemble various algorithms. For example, it is known how to certify a maximum matching in a bipartite graph [McC+11], and it is easy to check that a graph is k-colourable if the colouring of the graph is given. However, a subgraph isomorphism solver, such as the *Glasgow Subgraph Solver* [MPT20], may utilise both algorithms as subroutines: Maximum matching is used as part of all-different constraints, and a graph colouring can be used to obtain a bound on the size of a clique, as discussed in Paper H.

In this work, we propose a new, multi-purpose proof format based on pseudo-Boolean constraints (linear inequalities over 0-1 variables). The proof format builds on the cutting planes proof system [CCT87], theoretically studied in proof complexity, and extends it with rules to introduce new variables or eliminate certain assignments. We demonstrate how to use

this proof format to develop certifying algorithms for various problems that could not be certified efficiently with state-of-the-art techniques. This work focuses on combinatorial problems, including certifying implementations of SAT solving with parity reasoning and symmetry breaking, constraint programming, pseudo-Boolean optimisation, subgraph isomorphism, clique and maximum common subgraph solving.

Verified answers are achieved through a three-step process: In the first step, a pseudo-Boolean formula is constructed to describe the problem. Note that the satisfiability of pseudo-Boolean formulas is an NP-complete problem [Kar72], and hence we can use standard reductions to encode a wide range of problems into pseudo-Boolean formulas in polynomial time. Such an encoding might not always be practical, but for the studied applications, the encoding is relatively straightforward, and we are optimistic that this will be the case for a wide range of problems.

In the second step, the algorithm is run on the original problem but produces a proof for the pseudo-Boolean formula. The proof construction might require some extra bookkeeping to match the algorithm's internal state to the variables of the pseudo-Boolean problem. However, it turns out that for the studied algorithms, it is easy to express the made reasoning in the proof format, using information that is already available or very easy to obtain.

In the final step, we verify the constructed proof with the pseudo-Boolean formula using our verifier *VeriPB*[1]. The verifier can verify a solution, the solution's optimality, or that there are no solutions. In principle, the verifier can also verify the enumeration of solutions and translations between different pseudo-Boolean encodings, but the verified properties will vary based on the choice and use of rules.

This work contains two parts. The first part consists of six sections. The necessary preliminaries can be found in Section 2. Section 3 contains an introduction to the proposed proof system, Section 4 contains a worked-out example and Section 5 provides an overview of the included papers. Related work is discussed in Section 6 and Section 7 contains the concluding remarks. The second part consists of the papers included in this thesis.

[1] `https://gitlab.com/MIAOresearch/VeriPB`

2 Preliminaries

2.1 Basic Notation

For a natural number $n \in \mathbb{N}$ we use $[n]$ to denote the set $\{ 1, 2, \ldots, n \}$.

We assume that the reader is familiar with standard concepts of computational complexity, such as Turing machines, a language, the running time of an algorithm and asymptotic behaviour, and will only provide a terse description to refresh memory. A more extensive description can be found, for example, in [AB09].

The running time of an algorithm is measured as a function from the size of the input to the number of performed steps, where we use asymptotic notation $O(f(n)) = \{ g(n) \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n) \}$ as well as $\Omega(f(n)) = \{ g(n) \mid \exists c \in \mathbb{R}_+ : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n) \}$ where f and g are functions on natural numbers. A function $f(n)$ (such as the running time of an algorithm or Turing machine) is polynomial if there is a polynomial function $p(n)$ such that $f(n) \in O(p(n))$.

2.2 Graph Problems

A *graph* $G = (V, E)$ is a tuple consisting of a set of nodes V and edges $E \subseteq V \times V$. We will only consider undirected graphs, which means that if $(u, v) \in E$, then $(v, u) \in E$. The *degree* of a node v, denoted *degree*(v), is the number of edges incident to v. A graph has a *k-clique* if there is a subset of nodes $V' \subseteq V$ of size k such that there is an edge between any two nodes in V', i.e., $|V'| = k$ and for any $u, v \in V'$ with $u \neq v$ there is an edge $(u, v) \in E$. A *bipartite graph* $G = (V \cup V', E)$ is a graph with two disjoint sets of nodes V and V' and edges $E \subseteq V \times V' \cup V' \times V$. A *matching* on a bipartite graph is a subset of edges $E' \subseteq E$ such that each node is incident to at most one edge in E'. A maximum matching is a matching of maximal cardinality.

2.3 Formal Definition of Proof Systems

The concept of a propositional proof systems was introduced by Cook and Reckhow [CR79] to study the strength of different reasoning systems for propositional formulas, a research area known as *proof complexity*. This sec-

tion follows the standard definition as can be found, for example, in [BN21] with small adjustments to better accommodate the verification of the result of an algorithm.

An *alphabet* Σ is a finite set of symbols. A *language* L is a potentially infinite set of finite words x over an alphabet Σ. We use $|x|$ to denote the number of symbols in x. Generally speaking, a set of problems is just a language L and an answer is given in a different language L_A. We are interested in the language L_{CA} consisting of pairs of an input $x \in L$ and a correct answer $y \in L_A$, that is $x|y \in L_{CA}$ if and only if $x \in L, y \in L_A$ and y is considered a valid answer for the problem x. We use $|$ to denote a delimiter that does not appear in the alphabet of L or L_A and require that for $x|y \in L_{CA}$ the size of $x|y$ is polynomial in the size of x. A *proof system* for a language L_{CA} is a predicate $P(x, y, \pi)$ that has as inputs three words: a problem x, an answer y and a *proof* π (a proof may also be referred to as a *certificate*), and has the following three properties:

1. Soundness: If $x|y \notin L_{CA}$, then $P(x, y, \pi)$ is false for all π.
2. Completeness: If $x|y \in L_{CA}$, then there is a proof π such that $P(x, y, \pi)$ is true.
3. Polynomial-time verifiable: $P(x, y, \pi)$ can be computed (using a deterministic Turing machine) in time polynomial in the size of the inputs $|x|$,$|y|$ and $|\pi|$.

A *solver* for a language L is an algorithm that takes a word x over the alphabet of L, and if $x \in L$, then it produces an answer $y \in L_A$. If $x \notin L$, the solver does not produce an answer. A solver is correct within the language L if for all $x \in L$ the produced answer $y \in L_A$ is correct, i.e. $x|y \in L_{CA}$. A solver is *certifying* with respect to a proof system P if it also produces a proof π such that $P(x, y, \pi)$ is true. A certifying solver is also called a *proof logging* solver to emphasise that the proof is usually constructed during solving and not as a post-processing step.

The set **P** consists of all languages L that are efficiently solvable, i.e. that have a correct solver running in time polynomial in the size of the input (even if the input x is not in L). The set **NP** consists of languages L for which there is a proof system P such that for any word $x \in L$, there exists an answer $y \in L_A$ and a proof π, such that $|\pi| + |y|$ is polynomial in $|x|$ and $P(x, y, \pi)$ is true. A language $L \in \mathbf{NP}$ is *NP-complete* if for all $L' \in \mathbf{NP}$ there is a *polynomial-time reduction* from L' to L, i.e., a polynomial-time

algorithm A translating any word of L' into a word of L such that $x \in L'$ if and only if $A(x) \in L$.

2.4 Pseudo-Boolean Optimisation

In this section we introduce some standard notation used in pseudo-Boolean optimisation and related areas. A more detailed discussion can be found in [BN21].

A *Boolean variable* x can take values 0 (false) or 1 (true). A *literal* ℓ is either a variable x or its negation $\overline{x} = 1 - x$. We can also negate literals, where $\overline{\ell} = 1 - \ell$ and hence $\overline{\overline{x}} = x$. A *(linear) pseudo-Boolean constraint* C over literals $\ell_1, \dots \ell_n$ is a linear inequality over literals of the form

$$C\colon \sum_{i=1}^{n} a_i \ell_i \geq A, \tag{6}$$

where $A \in \mathbb{Z}$ is called the *defining constant* or *degree of falsity* and for all $i \in [n]$ the *coefficient* a_i is in $\mathbb{Z}$.

A pseudo-Boolean *formula* F is a set of pseudo-Boolean constraints. A *substitution* ρ is a (potentially partial) function from variables to 0, 1 or a literal. If a substitution ρ is not defined on a variable x, then we assume $\rho(x) = x$. A substitution is extended to literals in the canonical way, i.e. $\rho(\overline{x}) = 1 - \rho(x)$. A *(partial) assignment* is a substitution that only maps variables to 0, 1 or itself. For a partial assignment ρ, we use $\rho(\ell) = *$ to emphasise that ℓ is not assigned a value, i.e., $\rho(\ell) = \ell$. A *total assignment* is an assignment that assigns all variables to 0 or 1.

We can apply a substitution ρ to a pseudo-Boolean constraint C, denoted $C_{\restriction \rho}$, by applying the substitution to each literal individually, i.e.,

$$C_{\restriction \rho}\colon \sum_{i=1}^{n} a_i \cdot \rho(\ell_i) \geq A. \tag{7}$$

Note that for two substitutions ρ, ω it holds that $(C_{\restriction \omega})_{\restriction \rho}$ is the same as $C_{\restriction \rho \circ \omega}$, where $\rho \circ \omega(\ell) = \rho(\omega(\ell))$.

A total assignment ρ *satisfies* a constraint if $\sum_{\rho(\ell_i)=1} a_i \geq A$. A formula F is satisfied by a total assignment ρ if ρ satisfies all constraints in F. Such a substitution ρ is called a *solution* of F or a *satisfying assignment* of F. A

formula F is *satisfiable* if there is a satisfying assignment to F. A formula F *implies* a constraint C, written $F \models C$ if every satisfying assignment to F also satisfies C. A formula F implies another formula F' if $F \models C$ for all $C \in F'$.

It is always possible to transform a negative coefficient into a positive coefficient by flipping the sign of the literal. Consider a constraint of the form $-b\ell' + \sum_{i=1}^{n} a_i\ell_i \geq A$. By adding b to both sides of the inequality, we obtain $b - b\ell' + \sum_{i=1}^{n} a_i\ell_i \geq A + b$, which can be simplified to $b\overline{\ell'} + \sum_{i=1}^{n} a_i\ell_i \geq A + b$. Analogously, we can transform a negated variable into an unnegated variable by flipping the sign of the coefficient. Consider a constraint of the form $b\overline{y} + \sum_{i=1}^{n} a_i\ell_i \geq A$, which is the same as $b \cdot (1 - y) + \sum_{i=1}^{n} a_i\ell_i \geq A$ and can be simplified to $-by + \sum_{i=1}^{n} a_i\ell_i \geq A - b$.

A constraint that only has positive coefficients is in *(coefficient-)normalised form*, and a constraint that only has unnegated variables is in *variable-normalised form*. Any pseudo-Boolean constraint can be transformed into coefficient- or variable-normalised form as described above. It is easy to see that any assignment satisfying the constraint before the transformation also satisfies the constraint after the transformation. Therefore, we will consider a constraint to be the same no matter in which form it is. If not stated otherwise, we will think of constraints in coefficient-normalised form, but depending on the context, it can also be advantageous to think of constraints in their variable-normalised form. It can also be convenient not to have all literals on the left-hand side or to allow multiple occurrences of a literal. However such a constraint can always be brought into the form of (6) by rearranging terms, which does not change satisfying assignments. Additionally, we do not need to limit ourselves to greater-than-or-equal constraints as it is easy to see that an equality $\sum_{i=1}^{n} a_i\ell_i = A$ can be represented as two inequalities $\sum_{i=1}^{n} a_i\ell_i \geq A$ and $\sum_{i=1}^{n} -a_i\ell_i \geq -A$. Likewise, a greater-than constraint $\sum_{i=1}^{n} a_i\ell_i > A$ is the same as $\sum_{i=1}^{n} a_i\ell_i \geq A + 1$, because we only operate over integers.

The negation $\neg C$ of a pseudo-Boolean constraint $C\colon \sum_{i=1}^{n} a_i\ell_i \geq A$ is given by $\sum_{i=1}^{n} a_i\ell_i < A$, which can be rearranged to

$$\neg C\colon \sum_{i=1}^{n} -a_i\ell_i \geq -A + 1. \tag{8}$$

For a constraint $C\colon \sum_{i=1}^{n} a_i\ell_i \geq A$ in its coefficient-normalised form ($a_i \geq$

0) and a partial assignment ρ, we define

$$slack(\sum_{i=1}^{n} a_i\ell_i \geq A, \rho) = \sum_{i\in[n]:\rho(\ell_i)\neq 0} a_i - A. \tag{9}$$

If $slack(C, \rho) < 0$, then C cannot be satisfied by any extension of the assignment ρ and hence ρ falsifies C. If $slack(C, \rho) < a_i$ for some $i \in [n]$ such that $\rho(\ell_i) = *$, then C can only be satisfied by assigning ℓ_i to true and we say that C *propagates* ℓ_i. For a formula F and a (potentially empty) assignment ρ, *unit propagation* is the process of extending ρ by $\{ \ell \mapsto 1 \}$ if there is a constraint $C \in F$ that propagates ℓ until either ρ can no longer be extended through propagation or there is a constraint falsified by ρ. The latter is called a *conflict*.

For (linear) optimisation problems, we have an *objective function* of the form

$$f\colon \sum_{i=1}^{n} a_i\ell_i \tag{10}$$

with $a_i \in \mathbb{Z}$ for $i \in [n]$. We can apply a substitution ρ to an objective function f, denoted $f_{\restriction\rho}$, by applying the substitution to each literal individually as for constraints. Given an objective function f and a formula F, the goal of the optimisation problem

$$\min f \tag{11}$$

$$\text{such that } F \tag{12}$$

is to find a total assignment ρ satisfying F that is minimal with respect to f, i.e., there is no assignment ρ' such that ρ' satisfies F and $f_{\restriction\rho'} < f_{\restriction\rho}$. Note that it is sufficient to consider minimisation problems as every maximization problem can simply be expressed as a minimisation problem by negating the coefficients.

The answer for a pseudo-Boolean optimisation problem is either an optimal objective value $v \in \mathbb{Z}$ or $v = \mathit{UNSAT}$, stating that there is no satisfying assignment to F.

A *cardinality constraint* over literals $\ell_1, \ldots, \ell_n$ is a pseudo-Boolean constraint of the form

$$\sum_{i=1}^{n} \ell \geq A. \tag{13}$$

A *clause* is a disjunction ($\vee$, "or") of literals, which is the same as saying that at least one of the literals in the clause is true and hence a clause is the same as a cardinality constraint with $A = 1$. A formula in *conjunctive normal form (CNF)* is a pseudo-Boolean formula consisting only of clauses. The problem of finding a satisfying assignment to a CNF is also known as *SAT*, and a tool solving this problem is known as a SAT solver. We will say that some algorithm or encoding is *clausal* if it only operates on clauses. The *MaxSAT* problem is an optimisation variant of SAT, where the goal is to satisfy as many clauses as possible. *Constraint programming* is a more general formalism, in which we cannot only use pseudo-Boolean constraints and Boolean variables, but variables with larger domains and various kinds of constraints.

3 Description of the Proposed Proof System

Given a formula F and an objective function f, a proof for a pseudo-Boolean optimisation problem (f, F) with answer v consists of two parts. The first part is an assignment ρ^* satisfying F such that $f_{\restriction\rho^*} = v$ if $v \neq \mathit{UNSAT}$ or $\rho^* = \emptyset$ otherwise. The second part is a sequence of m pseudo-Boolean constraints $\pi = (D_1, D_2, \ldots, D_m)$ such that D_m is of the form $f \geq v$ if $v \neq \mathit{UNSAT}$ or $0 \geq 1$ otherwise. Furthermore, D_i is derived from $F \cup \{ D_1, \ldots, D_{i-1} \}$ using one of the rules below. To allow verification of the derivation in polynomial time, each step contains an annotation stating the used rule as well as additional information necessary for efficient verification. Each of these rules will preserve the value of an optimal assignment by guaranteeing that at least one optimal solution is left. This is stated more formally in the following invariant.

Invariant 1. If there is an optimal assignment ρ satisfying the formula $F \cup \{ D_1, \ldots, D_{i-1} \}$, then there is an assignment ρ' satisfying $F \cup \{ D_1, \ldots, D_i \}$ such that $f_{\restriction\rho'} = f_{\restriction\rho}$.

This invariant guarantees that whenever we are able to derive a constraint of the form $f \geq A$ for some $A \in \mathbb{N}$, then this constraint is a lower bound that must be satisfied by all optimal solutions to F. Note that the invariant holds vacuously if there is no solution to F, in which case adding any constraint is sound. Therefore, we will only discuss the case that there is an optimal solution ρ as described in the invariant. For most rules below, showing the invariant will be very simple: If not stated otherwise, we can choose $\rho' = \rho$ and the invariant follows trivially as ρ satisfies the premises of the rule and will satisfy the new constraint in an obvious way.

Let us now discuss the rules available in the proof system.

Literal Axioms. For each literal ℓ we can derive that the literal is at least zero

$$\frac{}{\ell \geq 0}\,. \tag{14}$$

Addition. We can add two constraints by adding their left-hand side and right-hand side respectively. When considering two constraints over variables $x_1, \ldots, x_n$ in their variable-normalised form (potentially with some zero coefficients) we get the rule

$$\frac{\sum_{i=1}^{n} a_i x_i \geq A \qquad \sum_{i=1}^{n} b_i x_i \geq B}{\sum_{i=1}^{n} (a_i + b_i) x_i \geq A + B}\,. \tag{15}$$

An addition step is annotated by the two constraints that are added. (In the actual proof format, each constraint is associated with a numeric identifier, and, instead of annotating a rule with a constraint, it is only annotated with the corresponding identifier.)

Multiplication. We can multiply a constraint by a positive integer $\alpha \in \mathbb{N}$.

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq A}{\sum_{i=1}^{n} \alpha \cdot a_i \ell_i \geq \alpha \cdot A} . \tag{16}$$

A multiplication step is annotated by the constraint multiplied and the used factor α.

Division. We can divide a constraint by a positive integer $\alpha \in \mathbb{N}$, if all coefficients a_i are divisible by α. The resulting constraint will only have integer values on the left-hand side and hence we can round up the right-hand side, which might not be an integer, resulting in the rule

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq A \qquad \forall i \in [n] : \alpha \text{ divides } a_i}{\sum_{i=1}^{n} (a_i/\alpha) \ell_i \geq \lceil A/\alpha \rceil} . \tag{17}$$

Note that by adding literal axioms, it is always possible to make coefficients divisible, without changing the degree. This is the same as rounding up the coefficients. To avoid ambiguities we will consider constraints in their coefficient-normalised form, resulting in the rule

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq A \qquad \forall i \in [n] : a_i \geq 0}{\sum_{i=1}^{n} \lceil a_i/\alpha \rceil \ell_i \geq \lceil A/\alpha \rceil} . \tag{18}$$

A division step is annotated by the divided constraint and the used divisor α. The proof system defined so far is known as *cutting planes* [CCT87] and is studied extensively in proof complexity (see [BN21] for an introduction).

Saturation. If we have a constraint with a coefficient that is larger than what is necessary to satisfy the constraint, no matter how the other literals are assigned, then we can reduce that coefficient. This is easiest to express for a constraint in its coefficient-normalised form, resulting in the rule

$$\frac{\sum_{i=1}^{n} a_i \ell_i \geq A \qquad \forall i \in [n] : a_i \geq 0}{\sum_{i=1}^{n} \min(A, a_i) \ell_i \geq A} . \tag{19}$$

A saturation step is annotated by the saturated constraint. The saturation rule is also used in pseudo-Boolean solvers based on [CK05].

Dominance-based strengthening (simplified). So far, all rules preserve solutions. That is, if an assignment satisfies the premises, then it also satisfies the conclusion. However, we only need to maintain optimal solutions, and hence we can add a constraint D_i that is falsified by non-optimal solutions. To check that a constraint only removes non-optimal solutions, we can verify that any satisfying assignment to F falsifying D_i can be mapped to another satisfying assignment of F with a better objective value. This mapping is defined through a substitution ω, and ρ is mapped to $\rho \circ \omega$, as we will see shortly. This results in the dominance rule

$$\frac{F \cup \{ \neg D_i \} \models F_{\restriction\omega} \cup \{ f > f_{\restriction\omega} \}}{D_i} . \tag{20}$$

If this rule holds and we have an assignment ρ satisfying F but falsifying D_i, then we know that ρ must satisfy the right hand side of the implication and hence $(F_{\restriction\omega} \cup \{ f > f_{\restriction\omega} \})_{\restriction\rho}$ is true. By rearranging the substitutions, we get that $F_{\restriction\rho\circ\omega} \cup \{ f_{\restriction\rho} > f_{\restriction\rho\circ\omega} \}$ must also be true. Hence, $\rho \circ \omega$ is a solution to F that has a better objective value than ρ.

A dominance step is annotated with the used substitution ω, and for each $C \in F_{\restriction\omega} \cup \{ f > f_{\restriction\omega} \}$ a proof that $F \cup \{ \neg D_i \} \cup \{ \neg C \}$ is unsatisfiable, from which the premises follows.

Invariant 1 holds by choosing $\rho' = \rho$, as an optimal assignment cannot falsify D_i: Let ρ be an optimal assignment as in Invariant 1. If ρ would falsify D_i, then, as discussed above, $\rho \circ \omega$ would be a solution to F with better objective value than ρ. This contradicts the fact that the ρ is an optimal assignment.

Redundancy-based strengthening (simplified). It is even possible to add a constraint D_i that is falsified by optimal solutions as long as not all optimal solutions are removed. To ensure that at least one optimal solution remains, let us consider an assignment ρ with the following property. The assignment ρ satisfies all previously added constraints

$$G_i = F \cup \{ D_1, \ldots, D_{i-1} \}, \tag{21}$$

but ρ falsifies the constraint D_i. Now assume that we can obtain another assignment $\rho' = \rho \circ \omega$ using a witness substitution ω such that ρ' satisfies G_i and D_i while also having an objective value that is at least as good as that of ρ. If we have such a witness ω, then clearly we didn't remove all optimal solutions. The redundancy rule is a formalization of this idea:

$$\frac{G_i \cup \{ \neg D_i \} \models (G_i \cup \{ D_i \})_{\restriction \omega} \cup \{ f \geq f_{\restriction \omega} \}}{D_i} . \tag{22}$$

A redundancy step is annotated with the used witness substitution ω, and for each $C \in (G_i \cup \{ D_i \})_{\restriction \omega} \cup \{ f \geq f_{\restriction \omega} \}$ a proof that $G_i \cup \{ \neg D_i \} \cup \{ \neg C \}$ is unsatisfiable, from which the premises follows. The redundancy rule is a generalization from the clausal redundancy rule [HKB17; BT19] to a pseudo-Boolean optimisation setting and can be used to introduce new variables (see Paper A).

Invariant 1 holds by choosing $\rho' = \rho$ if ρ satisfies D_i and otherwise $\rho' = \rho \circ \omega$. In the latter case, it is easy to see that we still have a satisfying assignment as ρ satisfies $G_i \cup \{ \neg D_i \}$ and hence $((G_i \cup \{ D_i \})_{\restriction \omega} \cup \{ f \geq f_{\restriction \omega} \})_{\restriction \rho}$ evaluates to true, which is the same as $(G_i \cup \{ D_i \})_{\restriction \rho'} \cup \{ f_{\restriction \rho} \geq f_{\restriction \rho'} \}$. Therefore ρ' is a solution with an objective value that is as least as good as an optimal solution and hence optimal itself.

Reverse Unit Propagation. It is often "obvious" that some constraint D_i follows from the current set of constraints, but annoying to write down an explicit proof. In such cases, it is useful to have methods for automatically proving that D_i can be derived, such as reverse unit propagation [GN03]. The idea of reverse unit propagation is that we perform unit propagation on the current set of constraints $G_i = F \cup \{ D_1, \ldots, D_{i-1} \}$ together with the negation of D_i. If the unit propagation results in a conflict, then we know that there is no assignment satisfying G_i but falsifying D_i, and hence every assignment satisfying G_i must also satisfy D_i. This gives the rule

$$\frac{F \cup \{ D_1, \ldots, D_{i-1} \} \cup \{ \neg D_i \} \text{ propagates to conflict}}{D_i} . \tag{23}$$

As unit propagation can be performed efficiently, it is sufficient to state the constraint D_i to be derived, and no further annotation is necessary. Invariant 1 holds choosing $\rho' = \rho$ because if $F \cup \{ D_1, \ldots, D_{i-1} \} \cup \{ \neg D_i \}$ propagates to conflict, then it is unsatisfiable and hence $F \cup \{ D_1, \ldots, D_{i-1} \} \models D_i$.

To show soundness of the proof system, assume for sake of contradiction that the given answer v is incorrect, but we have a valid proof. This means that if $v = \mathit{UNSAT}$, then there is an optimal solution ρ_0 to F, because we assumed that the answer is incorrect. If $v \neq \mathit{UNSAT}$, then either there is no solution to F or v is not the optimal value, and hence there is an

optimal solution ρ_0 with a different objective value. As the proof provides a solution ρ^* to F with $f_{\restriction \rho^*} = v$ it must be the case that there is an optimal solution ρ_0 with $f_{\restriction \rho_0} < v$.

Let ρ_0 be an assignment as described in the two cases above. Using Invariant 1 we can maintain a solution ρ_i to the current set of constraints $F \cup \{ D_1, \ldots, D_i \}$, and hence will end with the assignment ρ_m satisfying D_m. There are two cases to consider for ρ_m. If $v = \mathit{UNSAT}$, then D_m is $0 \geq 1$, and hence we reach a contradiction because this constraint is not satisfied by ρ_m. If $v \neq \mathit{UNSAT}$, then D_m is $f \geq v$, and hence we reach a contradiction because this constraint is not satisfied by ρ_m as $f_{\restriction \rho_m} \leq f_{\restriction \rho_0} < v$ by construction.

To show completeness of the proof system, we need to demonstrate that a valid proof can always be constructed for a correct answer. This follows directly from the fact that if $F \models C$, then C can be derived using cutting planes [Chv73] (this property is called *implicational completeness*). Therefore, if the formula F is unsatisfiable, then it holds that $F \models 0 \geq 1$ and $0 \geq 1$ can be derived. If the formula is satisfiable and has objective value v, then there is an optimal solution ρ^* with objective value v. As there is no better solution, it holds that $F \models f \geq v$, and hence $f \geq v$ can be derived.

For all rules, except for the dominance rule and the redundancy rule, it is easy to see that the rule can be verified in time polynomial in the size of the proof (including annotations). The dominance rule and the redundancy rule can contain another proof as annotation, and hence are recursive. Therefore we need an inductive argument. The proof (including annotations) is finite, and hence as a base case, we have a proof that does not contain the dominance or the redundancy rule, which can be checked in polynomial time. By induction, we know that all sub-proofs can be checked in polynomial time with respect to the size of the sub-proof. After checking that all necessary sub-proofs are provided and are valid, it is clear that the premise of the rule holds, and hence we are done checking the rule.

Some of the rules were simplified to provide a more gentle introduction to the proof system while preserving the main ideas. The full proof system, which also contains some additional rules, can be found in Paper B. There are three main differences: Firstly, the deletion rule is missing, which is important for efficient verification of the reverse unit propagation rule and memory efficiency. Secondly, the dominance rule does not have to improve the objective function, instead, we allow to define an arbitrary order on assignments to be improved. And finally, when a solution ρ to the current

set of constraints is found, then the constraint $f < f_{\restriction \rho}$ is added. Either this constraint enforces that the next solution needs to be better or this constraint can be used to derive contradiction, showing that no better solution exists.

4 A Worked-Out Example

Let us consider a simplified variant of a kidney exchange program. The idea is that a person, let us call him Frank, needs a kidney, and a friend of Frank, let us call her Sam, wants to donate a kidney to Frank. However, their blood type is not compatible, and hence Sam cannot donate her kidney to Frank directly. Maybe there is another pair of people (Susan and Max) having the same problem, and Sam's kidney is compatible with Susan, and Max's kidney is suitable for Frank, so they could make a cross-exchange. While such a pairing seems unlikely at first, if enough people register in a database for kidney exchanges, we will be able to find such pairings. This is the concept of kidney exchange programs as used in many countries.

The first step is to describe this problem in a formal model. Let us use a bipartite graph $G = (U \cup V, E)$ to represent the possible exchanges, as shown in Figure 2 (on the next page), where we use letters for donors on the left and numbers for recipients on the right. There is an edge between two nodes if a donated kidney is compatible with the recipient. For the example in Figure 2, donor A can donate a kidney to recipients 2 and 4. If a donor u donates his kidney to recipient v we will say that the edge (u, v) is *matched*. We want that each donor donates at most one kidney, and similarly, each recipient receives at most one kidney. This means we want a *matching*, i.e., a set of edges M such that each node is incident to at most one edge in M. Not having a working kidney is potentially life-threatening. Hence we want to find a maximum matching, i.e., a matching with as many matched edges as possible. The edges marked in Figure 2 form a maximum matching, but how can we ensure that there is no larger matching?

4.1 Verifying Optimality

Given that lives are at stake, we use a certifying algorithm to ensure that the found value is optimal. We start by focusing on the pseudo-Boolean formula and a proof of optimality that a matching algorithm could create. We will later discuss an algorithm producing this certificate, but for our initial purpose of verifying optimality, the inner workings of the algorithm are irrelevant.

To encode the matching problem into a pseudo-Boolean problem, we represent each edge $(u, v) \in E$ as Boolean variable $x_{u,v}$, which is true if and only if (u, v) is in the matching. We want to maximise the number of variables

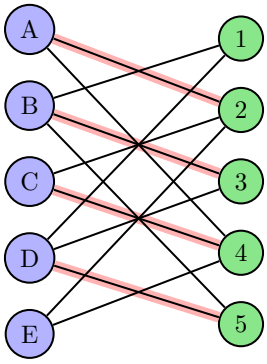


Figure 2: Maximum matching (highlighted in red) of a bipartite graph.

set to true, which is the same as minimising the variables set to false, i.e., $\sum_{(u,v)\in E} \overline{x}_{u,v}$. To enforce that no left node $u \in U$ has two matched edges, we add for every pair of edges $(u, v) \in E$ and $(u, v') \in E$ the constraint $\overline{x}_{u,v} + \overline{x}_{u,v'} \geq 1$. Analogously, we can enforce that no right node has two matched edges.

For the example in Figure 2 we have the objective function

$$\min : \sum_{i\in\{\,2,4\,\}} \overline{x}_{A,i} + \overline{x}_{C,i} + \overline{x}_{E,i} + \sum_{i\in\{\,1,3,5\,\}} \overline{x}_{B,i} + \overline{x}_{D,i} \tag{24}$$

and the formula

$$\begin{aligned}
\overline{x}_{A,2}+\overline{x}_{A,4} &\geq 1 && \text{(25a)} \\
\overline{x}_{B,1}+\overline{x}_{B,3} &\geq 1 && \text{(25b)} \\
\overline{x}_{B,1}+\overline{x}_{B,5} &\geq 1 && \text{(25c)} \\
\overline{x}_{B,3}+\overline{x}_{B,5} &\geq 1 && \text{(25d)} \\
\overline{x}_{C,2}+\overline{x}_{C,4} &\geq 1 && \text{(25e)} \\
\overline{x}_{D,1}+\overline{x}_{D,3} &\geq 1 && \text{(25f)} \\
\overline{x}_{D,1}+\overline{x}_{D,5} &\geq 1 && \text{(25g)} \\
\overline{x}_{D,3}+\overline{x}_{D,5} &\geq 1 && \text{(25h)} \\
\overline{x}_{E,2}+\overline{x}_{E,4} &\geq 1 && \text{(25i)} \\
\overline{x}_{B,1}+\overline{x}_{D,1} &\geq 1 && \text{(25j)} \\
\overline{x}_{A,2}+\overline{x}_{C,2} &\geq 1 && \text{(25k)} \\
\overline{x}_{A,2}+\overline{x}_{E,2} &\geq 1 && \text{(25l)} \\
\overline{x}_{C,2}+\overline{x}_{E,2} &\geq 1 && \text{(25m)} \\
\overline{x}_{B,3}+\overline{x}_{D,3} &\geq 1 && \text{(25n)} \\
\overline{x}_{A,3}+\overline{x}_{C,3} &\geq 1 && \text{(25o)} \\
\overline{x}_{A,3}+\overline{x}_{E,3} &\geq 1 && \text{(25p)} \\
\overline{x}_{C,3}+\overline{x}_{E,3} &\geq 1 && \text{(25q)} \\
\overline{x}_{B,5}+\overline{x}_{D,5} &\geq 1. && \text{(25r)}
\end{aligned}$$

At this point we should take a moment and double check that the pseudo-Boolean problem corresponds to the problem in the real world that we want to solve, because the choice of the pseudo-Boolean formula is not machine

verified. We need to convince ourself that any assignment satisfying this formula corresponds to a valid matching and vice versa. Note that in practice we would not only want that the formula encodes the matching problem represented by a graph as in Figure 2, but also that this graph is modelling the actual kidney donors and recipients that take part in the exchange program.

Let us now look at a proof that the matching in Figure 2 is optimal. The proof starts by using the addition rule with the constraints (25b)-(25d) to obtain the constraint $2\overline{x}_{B,1} + 2\overline{x}_{B,3} + 2\overline{x}_{B,5} \geq 3$. Applying the division rule with divisor 2 (and rounding up) yields

$$\overline{x}_{B,1} + \overline{x}_{B,3} + \overline{x}_{B,5} \geq 2. \tag{26}$$

This constraint says that all except one variable needs to be false, or put differently, at most one of the variables can be true (hence such a constraint is also known as an *at-most-one constraint*). We could have used this constraint directly to encode that each node should have at most one edge in the matching instead of encoding that no node should have two matched edges. However, this demonstrates that we can also use the proof system to re-encode the problem into something more convenient for proof logging. We can perform a similar derivation to obtain at-most-one constraints for the nodes D, 2 and 4, i.e.,

$$\overline{x}_{D,1} + \overline{x}_{D,3} + \overline{x}_{D,5} \geq 2 \tag{27}$$

$$\overline{x}_{A,2} + \overline{x}_{C,2} + \overline{x}_{E,2} \geq 2 \tag{28}$$

$$\overline{x}_{A,4} + \overline{x}_{C,4} + \overline{x}_{E,4} \geq 2. \tag{29}$$

By adding the constraints (26)-(29), we obtain

$$\sum_{i \in \{\,2,4\,\}} \overline{x}_{A,i} + \overline{x}_{C,i} + \overline{x}_{E,i} + \sum_{i \in \{\,1,3,5\,\}} \overline{x}_{B,i} + \overline{x}_{D,i} \geq 8, \tag{30}$$

which is a bound on the objective function (24).

The matching shown in Figure 2 assigns $x_{A,2}$, $x_{B,3}$, $x_{C,4}$ and $x_{D,5}$ to 1 and all other variables to 0. This assignment is part of the proof. To verify the proof, we first need to check that this assignment satisfies the constraints in the formula (25) and hence is a solution, which is an easy task to perform. Then we need to confirm that the objective value of the solution matches the derived lower bound. The objective function (24) contains only negated variables, and because 4 out of 12 variables in the objective function are set to true, the objective value is 8. We have a solution with an objective

value of 8 and a lower bound on the objective function (30) with the same value. Therefore, the solution (and hence the matching) must be optimal.

To check the derived lower bound (30), it suffices to verify the correct application of the rules in the proof system. Therefore, we do not need to check the proof manually but can use an automated tool such as *VeriPB* to take care of verification. All this verifier does is check each rule application individually. The verifier does not need to understand how the algorithm used to produce the matching works nor why it is correct. Furthermore, no knowledge about graph theory is needed to verify optimality.

This simplified variant of a kidney exchange program could already be verified with the methods described in [McC+11]. However, the advantage of our multi-purpose approach is that, in principle, it can deal with more complex formulations. For example, donor A might only be willing to donate a kidney if his friend recipient 1 receives a kidney, which can be enforced by adding the constraint $x_{A,2} + x_{A,3} \leq x_{B,1} + x_{D,1}$ to the formula.

4.2 Developing a Certifying Algorithm

Before we discuss an algorithm for finding a maximum matching and how we can construct a proof, we need some basics from graph theory. Let $G = (U \cup V, E)$ be a bipartite graph and let $M \subseteq E$ be a matching. We will say that a node u is *matched* if u is incident to an edge in the matching M. If a node is not matched, it is *unmatched*. An *alternating path* is a sequence of nodes $p = (v_1, \ldots, v_k)$ of length $k \in \mathbb{N}$ such that v_1 is unmatched and every other edge along the path is in the matching, i.e., for all even $i \in [k]$ the edge (v_{i-1}, v_i) is in E but not in the matching M and (v_i, v_{i+1}) is in M unless $i = k$. We use $\mathit{edges}(p)$ to denote the set of edges along the path p. An *augmenting path* is an alternating path that ends with an unmatched node. If there is an augmenting path p then the matching M is not optimal, as we can construct a larger matching by flipping whether an edge along the path p is in the matching, i.e., $M' = M \setminus \mathit{edges}(p) \cup \mathit{edges}(p) \setminus M$. This changes v_1 and v_k from unmatched nodes to matched nodes while maintaining the status for all other nodes and increasing the size of the matching by 1.

Consider the following standard algorithm. We start with the empty matching and then greedily increase the size of the matching by finding augmenting paths starting at unmatched nodes in U. If there is no further augmenting path, the algorithm terminates and returns the last matching. For

finding an augmenting path, the algorithm computes the set R of nodes reachable via an alternating path starting at an unmatched node in U (for example, via a modified breath-first search).

To provide proof logging for this algorithm, we consider the set $W = (U \setminus R) \cup (V \cap R)$. For each node u in W we derive an at-most-one constraint over the variables corresponding to its incident edges. Then we use the addition rule to add all of these at-most-one constraints together. Additionally, we add literal axioms $\overline{x}_{u,v} \geq 0$ for any $u, v \in W$ with an edge $(u, v) \in E$. We will discuss later why this results in a lower bound to the objective function matching the found solution. Finally, the algorithm writes the assignment to the Boolean variables corresponding to the found matching into the proof file.

For the example in Figure 2, the only unmatched node in U is E, and nodes reachable via an alternating path are $R = \{ A, C, E, 2, 4 \}$. There is no node $v \in V$ on the right that is reachable and unmatched, and hence the algorithm does not find an augmenting path and terminates. For proof logging, it computes $W = \{ B, D \} \cup \{ 2, 4 \}$. The at-most-one constraints to be derived and added together are (26)-(29). There are no edges between nodes in W, and hence the derivation of the objective bound is completed. The solution written to the proof file assigns $x_{A,2}$, $x_{B,3}$, $x_{C,4}$ and $x_{D,5}$ to 1 and all other variables to 0.

With this description, it is possible to implement the algorithm, run it and verify the produced proof to check if the result is correct. Somewhat surprisingly, this means that we can verify that a result of the algorithm is valid on some specific input without even knowing why this algorithm is correct.

Now, let us discuss why the described algorithm and proof logging will always produce a valid bound on the objective. For this, we need Kőnigs theorem, stating that for any maximum matching M in a bipartite graph, there is a *vertex-cover* W, i.e., a set of nodes such that every edge in the graph is incident to a node in W, such that $|M| = |W|$. For each node in the graph we can construct an at-most-one constraint over the variables corresponding to the incident edges. If we add the at-most-one constraint of each node in a vertex cover W, then we get a constraint stating that at most $|W|$ of the variables in the constraint are true. Because W is a vertex cover, every variable must appear at least once in the constraint. If a variable appears more than once, we can add the literal axiom $x_{u,v} \geq 0$, which is the same as $-x_{u,v} \leq 0$, to reduce the variable's coefficient to 1

while maintaining an at-most-$|W|$ constraint. The result is a constraint that gives a bound on the objective function with the same value as the size of the maximum matching found.

The set $W = (U \setminus R) \cup (V \cap R)$ used for proof logging in the algorithm is the same set used to prove Kőnigs theorem. We only have to show that W is indeed a *vertex-cover* with the same size as the found matching M. Let us consider an edge $(u, v) \in E$ with $u \in U$ and $v \in V$. We first observe that if u is in R then v is in R: Indeed, if u is in R and (u, v) is in the matching, then u is matched and can only have been reached from its matched neighbour v. And if (u, v) is not in the matching, then we can reach v from u by extending the alternating path that reached u. Using this observation, we can show that for every edge $(u, v) \in E$ with $u \in U$ and $v \in V$ either u or v is in W: On the one hand, if $u \notin W$, then $u \in R$ and as observed, it follows that $v \in R$ and hence $v \in W$. On the other hand, if $v \notin W$, then $v \notin R$ and thus $u \notin R$ by the observation above, and we have $u \in W$. Therefore, W is a vertex cover. Clearly $|M| \leq |W|$, because W is a vertex cover and hence we have at least one node of each matched edge in W. Furthermore, $U \setminus R$ only contains matched nodes because every unmatched node in U is contained in R and $V \cap R$ only contains matched nodes, or there would be an augmenting path and the algorithm would not have terminated. There cannot be an edge (u, v) in the matching M with both endpoints in the vertex cover, i.e., $u \in U \setminus R$ and $v \in V \cap R$, because then u is reachable by extending the alternating path reaching v, and u would be in R. Therefore, we have a different matched edge for each node in W and hence $|W| \leq |M|$.

5 Main Results of the Research Papers

This section gives a short overview of the papers included in this thesis. The proof system described in the Section 3 was developed and improved as an ongoing effort while studying different applications. As Section 3 already contains an overview of the rules in the proof system, this section focuses on the applications studied.

The included papers can roughly be divided into four groups. The first group focuses on important techniques in SAT and MaxSAT solving that are challenging to certify efficiently or cannot be certified using the de-facto standard *DRAT*. Paper A investigates parity reasoning and Paper B studies symmetry breaking in SAT solving. Paper D studies the translation of pseudo-Boolean problems into SAT problems, and in Paper C we take a first step towards proof logging techniques used in MaxSAT, which is an optimization version of SAT, by lifting MaxSAT techniques to a pseudo-Boolean setting. The second group of papers studies constraint programming, an area that does not have an established proof format. While basic search algorithms in constraint programming are usually similar to SAT solving and hence proof logging techniques are transferable, the challenge is providing proof logging for the various algorithms for propagating constraints. An important constraint is the all-different constraint for which we study proof logging in Paper E, and in Paper F we discuss proof logging for a wider range of basic constraints. The third group of papers studies proof logging for different graph problems demonstrating the versatility of the proposed proof format. Paper G investigates subgraph isomorphism, and Paper H studies maximum clique as well as maximum common subgraph. Finally, the last group, containing only Paper I, is not an application of proof logging but studies the division and saturation rule in more depth.

5.1 Summary of Paper A

"Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs"

A parity or XOR-constraint is a linear equality modulo 2. For a set of parity constraints it is possible to efficiently determine satisfiability, as well as propagation of variables, via systematic addition of parity constraints through Gaussian elimination. This is crucial for the performance of SAT solvers on instances that encode such parity constraints into CNF. It was

observed in [DGP04] that with the right encoding, addition of parity constraints can be performed using cutting planes. In Paper A we use this observation to provide full proof logging for parity reasoning by first translating a standard CNF encoding of parities into a pseudo-Boolean encoding. With this encoding we can easily certify the addition of parities, which is used to derive parities that cause a propagation. As the SAT solver only understands clauses, we also describe how to derive a clause that has the same propagation as a given parity constraint.

An important step to allow for the translation from the CNF encoding into the pseudo-Boolean encoding was adding the redundancy-based strengthening rule to the proof system, which allows for the introduction of auxiliary variables. With this rule our proof format is a generalisation of *DRAT* and, therefore, allows reusing of previously existing proof logging for *DRAT*, such as SAT pre- and in-processing.

The evaluation demonstrates a dramatic improvement in proof logging and verification performance compared to previous work, certifying parity reasoning using *DRAT*.

To improve the efficiency of a SAT solver with parity reasoning, clauses can be constructed lazily [SGM20], i.e., only when they are needed in the solver. The same technique improves proof logging and verification performance by producing certificates only for clauses that are needed.

5.2 Summary of Paper B

"Certified Symmetry and Dominance Breaking for Combinatorial Optimisation"

A formula F has a *syntactic symmetry* if we can permute literals without changing the formula, i.e., if there is a substitution ρ that only permutes literals such that $F_{\restriction\rho} = F$. Symmetry breaking adds constraints to the formula that are falsified by most but not all assignments that are symmetric. This technique can improve a solver's performance drastically by eliminating symmetric parts of the search space while guaranteeing correctness as at least one assignment is left in each equivalence class of symmetric assignments.

Symmetry breaking was studied previously for *DRAT* [HHW15], but the approach can require iterating over the broken symmetries multiple times,

with each iteration introducing a linear number of variables. Thus the method is only practical for small symmetries that can interact in simple ways. With our proof system, we can consider multiple symmetries independently using the dominance-based strengthening rule introduced in Paper B. In contrast to the approach for *DRAT*, the pseudo-Boolean symmetry breaking constraints can be added without introducing new variables. An evaluation of SAT competition benchmarks demonstrates the practicability of our system.

As described in Paper B, this rule can be used for symmetry breaking for CNF formulas and constraint programming as well as dominance breaking for finding a maximum clique. A surprising observation in this paper is that, carefully defining the deletion rule becomes crucial when the dominance rule is used. If deletion were possible without limitation, then the proof system would become unsound and allow to derive a contradicting constraint from a satisfiable formula.

5.3 Summary of Paper C

"Cutting to the Core of Pseudo-Boolean Optimization: Combining Core-Guided Search with Cutting Planes Reasoning"

As the proof format can handle optimisation problems, it is a natural question to ask if it can be used to design certifying MaxSAT algorithms. While encoding a MaxSAT problem into a pseudo-Boolean problem is straightforward, it is more challenging to translate the algorithms from one formalism to the other. Doing so, however, is not only a step towards proof logging for MaxSAT but also opens the possibility to explore a generalization of these algorithms in a pseudo-Boolean setting.

In Paper C we translate a technique known as core-guided solving from MaxSAT to a pseudo-Boolean setting and implement it into the solver *RoundingSat* [RS]. Interestingly, using a pseudo-Boolean solver instead of a SAT solver as a subroutine has multiple advantages: Firstly, it can use cardinality constraints directly and does not require re-encoding. Secondly, the solver subroutine can return not only a clause as an answer but also a general pseudo-Boolean constraint. These constraints allow stronger bounds on the objective function and improve solver performance. The evaluation demonstrates that we can significantly improve the performance of *RoundingSat* on optimisation instances.

5.4 Summary of Paper D

"Certified CNF Translations for Pseudo-Boolean Solving"

A crucial technique in SAT-based pseudo-Boolean and MaxSAT solving is translating pseudo-Boolean constraints into clausal form. In Paper D, we describe how to certify the translation to three different encodings. A significant challenge is that there are many different encodings, making it laborious to provide proof logging for all of them. We develop a unifying framework to streamline proof logging for a wide range of encodings. The framework utilises the fact that many encodings can be viewed as a CNF encoding of a circuit computing an integer representation of the sum of the variables in the constraint. To support different encodings for the integer representation, we provide proof logging for a set of basic building blocks that can then be put together to provide proof logging for different encodings.

To demonstrate the approach, we combine our tool for translating pseudo-Boolean constraints with a state-of-the-art SAT solver into a fully certified pseudo-Boolean solver. This results in the first SAT-based pseudo-Boolean solver that supports verification of the full workflow instead of just verifying the SAT solver and trusting the translation. The evaluation demonstrates that the approach is viable for most problems in the last pseudo-Boolean competition [PB16], although there is still room for improvement in the asymptotic behaviour on large constraints.

5.5 Summary of Paper E

"Justifying All Differences Using Pseudo-Boolean Reasoning"

Given a set of variables V and for each variable $X \in V$ a set of values $\mathit{domain}(X)$ that this variable can take, the all-different constraint enforces that each value in $\bigcup_{X \in V} \mathit{domain}(X)$ is assigned to at most one variable in V. In Paper E we provide proof logging for the all-different constraint to evaluate if our proof format can also be used to design certifying algorithms that do not operate on Boolean variables. Studying propagation for the all-different constraint is especially interesting as it has been a challenging barrier to providing proof logging for constraint programming. Most state-of-the-art all-different propagators are based on finding an optimal matching in a bipartite graph, where the variables correspond to the left

and the values to the right partition, and there is an edge if a value can be assigned to a variable. A matching in this graph corresponds to a valid assignment of variables to values. While certifying algorithms for matching have been studied in [McC+11], they use problem-specific certificates, and it is unclear if and how such certificates could be integrated into a larger system that is using matching only as a subroutine, as is the case in a constraint programming solver.

Our proof format allows certifying maximum matchings, and it is easy to integrate the resulting certificate into a larger system such as a constraint programming solver. An important insight of Paper E is that having a reverse unit propagation rule is crucial for ease of use. Reverse unit propagation reduces the burden on the implementer of the solver to produce a proof manually: Instead of carefully constructing a derivation, it is sufficient to claim that a certain constraint is implied, as long as it can be checked automatically, i.e., using reverse unit propagation.

A key benefit of our proof system is that the resulting proof logging is much simpler and more efficient than what would be possible with a clausal proof system: Note that matching is a generalisation of a problem known as the pigeonhole principle. The pigeonhole principle requires exponential sized [Hak85] *resolution proofs*, which is a simple clausal proof system. Hence proof logging in a clausal format would require more sophisticated techniques, like introducing auxiliary variables, which might result in prohibitive overhead.

5.6 Summary of Paper F

"An Auditable Constraint Programming Solver"

In Paper E we already provided proof logging for the all-different constraint. In Paper F we extend this to a full constraint programming solver with support for bounded integer variables and a reasonable range of constraints such as linear inequalities, table constraints and (2D) element constraints. From a proof logging perspective, an obstacle is that different encodings are desirable for the same variable depending on the constraints. This is resolved by encoding the problem into a pseudo-Boolean formula using one encoding and then introducing the other encodings on demand via rules in the proof system. With the implemented constraint programming solver we demonstrate that, with some careful design choices, it is reasonably simple to implement proof logging for constraint programming.

One limitation of this approach is that we cannot verify the correctness of the specification of the problem given as pseudo-Boolean formula. Instead, we need to rely on extensive testing of the encoding on a constraint by constraint basis.

5.7 Summary of Paper G

"Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions"

Preliminaries. Given a pattern graph G_1 and a target graph G_2 the *(non-induced) subgraph isomorphism problem* is to find a total injective mapping π from nodes in G_1 to nodes in G_2 such that edges in G_1 map to edges in G_2.

Paper G shows that in hindsight it is not too complicated to provide certification for a state-of-the-art subgraph isomorphism solver. This is surprising as our proof format does not know what a graph is, let alone does it understand any concepts from graph theory such as the degree of a node or the neighbourhood of a node. However, the studied subgraph isomorphism solver heavily relies on these concepts. For example, a pattern node u cannot be matched to a target node v if the degree of node u is larger than the degree of node v. Another example is that if there are $k \in \mathbb{N}$ paths from a node u to a node v in the pattern graph, then there needs to be at least k paths between $\pi(v)$ and $\pi(u)$ in the target graph.

5.8 Summary of Paper H

"Certifying Solvers for Clique and Maximum Common (Connected) Subgraph Problems"

Preliminaries. For two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a *common (induced) subgraph* is implicitly given through a partial injective mapping π from V_1 to V_2 such that edges in G_1 map to edges in G_2 and non-edges map to non-edges, i.e., if π is defined for u and v then $(u, v) \in E_1$ if and only if $(\pi(u), \pi(v)) \in E_2$. For the maximum common induced subgraph problem, the goal is to map as many nodes as possible. A common induced subgraph is connected if all mapped nodes in G_1 (and hence in G_2) are connected through a sequence of edges. A graph $G = (V, E)$ is *k-colourable*

if there is a way to colour each node with a colour such that at most k colours are used, and if there is an edge $(u, v) \in E$ then u and v have a different colour.

In Paper H we provide proof logging for the optimisation problems maximum clique and maximum (common) subgraph.

An essential technique for solving maximum clique, that we can now certify, is to derive an upper bound on the size of a clique using a graph colouring: If a graph is k-colourable, then it cannot contain a $(k+1)$-clique, as each member of a clique needs to get a different colour. A fascinating insight is that we can derive a valid bound in our proof system from any valid colouring, and hence it does not matter how the graph colouring is obtained. Therefore, if the graph colouring algorithm changes, the proof logging does not need to be adjusted.

Additionally, Paper H studies proof logging for two different approaches for solving the maximum common subgraph problem, and depending on which approach is used, a different pseudo-Boolean specification of the problem is desirable for proof logging. The first approach is to find a mapping between the graphs directly. The second approach is to reduce the problem to clique, that is, to construct a graph such that a clique in the graph corresponds to a solution of the subgraph isomorphism problem and vice versa. After reducing the problem to clique, a clique solver and its proof logging can be used without modification. For this to be possible, it is necessary to have a different pseudo-Boolean formula, depending on the chosen algorithm. However, a specification should be chosen based on its simplicity and should not vary based on the used algorithm. So, instead of using different pseudo-Boolean formulas based on the used algorithm, the paper describes how to derive all constraints needed for proof logging the clique algorithm from the specification of the pseudo-Boolean encoding of the subgraph isomorphism problem.

Another challenge solved in Paper H is that certifying a maximum common *connected* subgraph requires defining connectedness. However, our proof system does not understand any graph theory. Instead, we add constraints to the pseudo-Boolean formula that are designed to only be satisfied by assignments of the Boolean variables that correspond to a connected subgraph.

5.9 Summary of Paper I

"On Division Versus Saturation in Pseudo-Boolean Solving"

An important consideration when designing a proof format is which rules should be available. The goal is, on the one hand, to have as few and as simple rules as possible to make the verifier easier to implement and more trustworthy. On the other hand, we want the format to be expressive enough, such that constructing concise proofs is easy for the solver. These two conflicting goals become already relevant in the context of pseudo-Boolean solvers that reason with pseudo-Boolean constraints natively. Such solvers can use either the saturation rule or the division rule, which raises the question if the proof format should support both of these rules or if it is sufficient to support only one. In Paper I we study the power of division and saturation from a proof complexity point of view, where we ask what are the shortest refutations for an unsatisfiable formula. In [Vin+18] it was already shown that division is stronger than saturation on constraints with coefficients of polynomial magnitude, and we were able to generalize this result to any cutting planes derivation with saturation, no matter the size of the coefficients involved.

Proposition 1. There is a family of PB formulas $\{ F_n \}_{n \in \mathbb{N}_+}$ with $O(n)$ variables and constraints that can be refuted in length $O(n)$ in cutting planes with division, but for which any saturation refutation has length $\Omega(\exp(n))$.

This means that the saturation rule cannot be used to replace the division rule in a proof system. While it was shown in [Vin+18] that saturation can be replaced with multiple division steps, we show that the number of division steps required to replace a single saturation depends on the size of the saturated coefficient.

Proposition 2. From $2Rx + \sum_{i=1}^{2R} z_i \geq R$ saturation can derive $Rx + \sum_{i=1}^{2R} z_i \geq R$ in one step, while any cutting planes derivation with division requires at least $\Omega(\sqrt{R})$ steps.

Note that R is exponential in the bit-size used to represent the coefficient R in the constraint, and hence replacing a single saturation step can require a large number of division steps. While this result is not as strong as the previous result, in particular because the constraint already contains R

variables, it is clear that it is desirable to also include the saturation rule as part of a practical proof system.

These results are not only relevant for proof logging but also for solver design, where a choice is made between using the saturation or division rule. Therefore Paper I is also emphasising another aspect of the rules: Pseudo-Boolean solvers such as *RoundingSat* and *Sat4j* [SAT4j] will only perform the addition of constraints C and D if there is a literal ℓ that appears in C and appears negated in D. Before the addition, the constraints are multiplied such that the resulting constraint no longer contains ℓ. The presented results also translate to this restricted form of cutting planes, and hence we are able to present formulas that should in principle be able to discriminate between solvers that only use division or only use saturation. However, these formulas were hard for both kinds of solvers, which indicates that further improvements to the used heuristics are necessary.

6 Related Work

There are various proof formats that target a single application. The practical algebraic calculus (*PAC*) [RBK18; KFB20; Kau+22] operates on polynomial equations. It is designed to verify algebraic circuits, e.g., for multiplication, and allows the integration of basic SAT solving techniques [KBK20]. There is a proof format for basic integer programming techniques [CGS17] supporting addition and rounding as well as branch and bound, but lacking support for more advanced techniques such as preprocessing. For basic constraint programming techniques, a format was proposed in [VS10] with the intention that it can be extended by additional rules for each used algorithm or propagator. There are numerous proof logging formats for SAT such as *RUP* [GN03], *TraceCheck* [Bie06], *DRAT* [HHW13a; HHW13b; WHH14], *GRIT* [CMS17], *LRAT* [Cru+17], and *FRAT* [BCH21]. All of the mentioned formats were specifically designed for a single application, such as SAT solving. However, it is conceivable that some formats, such as *DRAT*, could also be utilised as a multi-purpose proof format.

There are two concerns when designing a multi-purpose proof format. The first concern is how the problem to be solved is specified. The CNF input format used for SAT solving is not suited to describe optimisation problems. Consequently, a proof format designed for CNF, such as *DRAT*, cannot verify optimality. Our approach builds on pseudo-Boolean optimisation and hence can deal with optimisation, but it would be an ill fit for quantified Boolean formulas. The second concern is how to express the reasoning of a wide range of algorithms within the rules of the proof system.

There are two main ideas for supporting various algorithms. The first idea is to make the proof format extensible [VS10; BCH21], i.e., to allow the addition of new rules based on the used algorithms. While this approach is very flexible, it increases the complexity of the verifier with each new rule, and there might be subtle interactions between different rules threatening the soundness of the proof format. The second idea is to design a proof system that is powerful enough outright. From a theoretical perspective, this is surprisingly easy: By allowing the introduction of new variables, proof systems such as cutting planes and resolution become as powerful as any extended Frege proof system, i.e., a proof system having an arbitrary but finite number of rules over Boolean circuits. This approach is followed by many proof systems such as *DRAT*, *PAC* and our proof system. However, applying this theoretical result in practice can result in

a considerable polynomial overhead. This overhead can be ignored from a theoretical perspective when studying super polynomial differences in proof size, but already a small polynomial overhead can quickly become prohibitive in practice. Therefore, it is crucial for a multi-purpose proof format that simple problems are easy to express, while more sophisticated algorithms can also exhibit more sophisticated proof logging. In this spirit, an argument for choosing cutting planes as the basis of our proof system is that refuting the pigeonhole principle, asking if $n \in \mathbb{N}$ pigeons can fit into $n-1$ holes, is easy in cutting planes but requires proofs of exponential size in resolution [Hak85] and polynomial calculus [Raz98; IPS99]. Resolution can be seen as the foundation of clausal proof systems such as *DRAT*, and respectively polynomial calculus is the foundation of *PAC*. Hence, refuting a problem as simple as the pigeonhole principle in these proof systems requires some sophistication already, while it is straightforward in our proof system.

7 Discussion and Conclusion

This thesis proposes a new proof system based on pseudo-Boolean constraints and the cutting planes proof system to certify the answer of combinatorial algorithms. To the best of our knowledge, it is the first multi-purpose proof system, certifying a wide range of different problems and algorithms. The result is the verifier *VeriPB*, which can verify SAT and pseudo-Boolean problems as well as constraint programming and graph problems such as cliques, subgraph isomorphisms, and common connected subgraphs. The focus of the proof system is to verify the optimality of solutions or that no solution exists. Still, it can, in principle, be used for other tasks, such as enumerating solutions or verifying the translation between different pseudo-Boolean encodings.

With the help of this new proof system, we can design and implement certifying versions of multiple algorithms for which no such implementation existed or previous certifying algorithms were only practical for simple instances. Importantly, adding proof logging to the algorithms does not require substantial changes. The information already available to the algorithm is sufficient to construct the necessary proofs.

We achieved the goal of not blindly trusting the solver by using certifying algorithms, but this raises the question if we should trust the verifier more than the solver. The proof system checked by the verifier only has a few rules that are easy to verify and it should be easier to implement the verifier correctly than the solver. Although this is left for future work, it is within reach and currently work in progress to produce a formally verified version of the verifier, as is done for other proof formats [CMS17; Cru+17; Kau+22]. However, even if we choose not to believe that the verifier is always correct, we can trust a solution more after verification: On the one hand, it is less likely that the verifier and the solver have a bug for the same problem. On the other hand, the verifier will be thoroughly tested over time as the same verifier can verify multiple, completely different algorithms through the multi-purpose nature of the proof format. Therefore, fixing a bug in the verifier that was found for one of the applications also benefits the others.

It would be desirable to verify the solvers formally. However, this still seems to be out of reach for complex and efficient solvers, with the notable exception of *IsaSAT* [Bla+18; Fle19]. The reason is that formal verification usually proves that the solver is producing a correct answer on all possible

inputs. Therefore, it is necessary to formalise and prove the correctness of all used data structures and algorithms. With certifying algorithms, on the other hand, the verifier only verifies that an answer to a single problem is correct. Notably, the verifier can prove the correctness of the answer without knowing the underlying data structures and algorithms, let alone why these algorithms are correct. For example, the algorithm for all-different constraints utilises graph theory results, such as finding a maximum matching via the alternating paths algorithm. However, the verifier does not even know what a graph is nor why the alternating paths algorithm produces a maximum matching. Instead, it only works on a simple pseudo-Boolean representation with a few rules, which is sufficient to verify the correctness of the answer provided by the solver. Additionally, even for formally verified solvers, it could be beneficial to use proof logging as this could detect hardware bugs and failures.

The overhead in running time for proof logging and verification can be a limiting factor to the success of certifying algorithms. The observed overhead varies widely between being negligible to orders of magnitude, depending on the studied applications and concrete problem instances. For some use cases, this is not necessarily a problem. For example, verifying problems with easy to medium difficulty can already help to find bugs during development. However, improved performance would be desirable. The first concern is the amount of data that must be written to disk for the pseudo-Boolean formula and the proof file. The file size could be improved through on-the-fly compression, which reduces file size but increases computational cost, or by using a binary format instead of a textual format, which has been used successfully for *DRAT-trim* [HHW13a]. Furthermore, the file size of the proof file can be reduced by only producing proof logging for necessary constraints, as is done, for example, in Paper A. However, it might not be possible or it might require additional book keeping for the solver to determine apriori which constraints will be used. An alternative approach is to trim the proof afterwards, as is done in *DRAT-trim*. While this has no effect on the overhead of proof logging, it can reduce the time spent verifying a proof. Finally, it could be interesting to investigate if and how verification can be parallelised to utilise multiprocessor architectures.

For future work, besides providing a formally verified verifier and improving the performance of proof logging and of the verifier, it would be desirable to establish proof logging for more algorithms. Another interesting use case of certifying algorithms that could be investigated is to utilise the generated proofs to analyse and improve solvers as in [Elf+18; SKM19].

8 References

[AB09] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. 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)*. Vol. 12651. Lecture Notes in Computer Science. 2021, pp. 59–75.

[Bie06] Armin Biere. *TraceCheck*. `http://fmv.jku.at/tracecheck/`. 2006.

[Bla+18] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich and Christoph Weidenbach. "A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality". In: *Journal of Automated Reasoning* 61 (2018), pp. 333–365.

[BN21] Samuel R. Buss and Jakob Nordström. "Proof Complexity and SAT Solving". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn J. H. Heule, Hans van Maaren and Toby Walsh. 2nd. Vol. 336. Frontiers in Artificial Intelligence and Applications. 2021. Chap. 7, pp. 233–350.

[Bog+22] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Certified Symmetry and Dominance Breaking for Combinatorial Optimisation". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '22)*. Vol. 36. 2022.

[BT19] Samuel R. Buss and Neil Thapen. "DRAT Proofs, Propagation Redundancy, and Extended Resolution". In: *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*. Vol. 11628. Lecture Notes in Computer Science. 2019, pp. 71–89.

[CCT87] William Cook, Collette Rene Coullard and György Turán. "On the Complexity of Cutting-Plane Proofs". In: *Discrete Applied Mathematics* 18.1 (1987), pp. 25–38.

[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)*. Vol. 10328. Lecture Notes in Computer Science. 2017, pp. 148–160.

[Chv73] Vašek Chvátal. “Edmonds polytopes and a hierarchy of combinatorial problems”. In: *Discrete Mathematics* 4.4 (1973), pp. 305–337.

[CK05] Donald Chai and Andreas Kuehlmann. “A Fast Pseudo-Boolean Constraint Solver”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.3 (2005). Preliminary version in *DAC ’03*, pp. 305–317.

[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)*. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 118–135.

[CR79] Stephen A. Cook and Robert A. Reckhow. “The Relative Efficiency of Propositional Proof Systems”. In: *Journal of Symbolic Logic* 44.1 (1979). Preliminary version in *STOC ’74*, pp. 36–50.

[Cru+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)*. Vol. 10395. Lecture Notes in Computer Science. 2017, pp. 220–236.

[Dev+16] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe and Marc Denecker. “Improved Static Symmetry Breaking for SAT”. In: *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT ’16)*. Vol. 9710. Lecture Notes in Computer Science. 2016, pp. 104–122.

[Dev+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 AAAI Conference on Artificial Intelligence (AAAI ’21)*. Vol. 35. 5. 2021, pp. 3768–3777.

[DGP04] Heidi E. Dixon, Matthew L. Ginsberg and Andrew J. Parkes. “Generalizing Boolean Satisfiability I: Background and Survey of Existing Work”. In: *Journal of Artificial Intelligence Research* 21 (2004), pp. 193–243.

[Elf+18] Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström and Laurent Simon. "Seeking Practical CDCL Insights from Theoretical SAT Benchmarks." In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, (IJCAI '18)*. 2018, pp. 1300–1308.

[Elf+20] Jan Elffers, Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Justifying All Differences Using Pseudo-Boolean Reasoning". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '20)*. Vol. 34. 02. 2020, pp. 1486–1494.

[Fle19] Mathias Fleury. "Optimizing a verified SAT solver". In: *Proceedings of the NASA Formal Methods Symposium*. Vol. 11460. Lecture Notes in Computer Science. 2019, pp. 148–165.

[GMN20] Stephan Gocht, Ciaran McCreesh and Jakob Nordström. "Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions". In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, (IJCAI '20)*. 2020, pp. 1134–1140.

[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)*. 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)*. 2003, pp. 886–891.

[GN21] Stephan Gocht and Jakob Nordström. "Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs". In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '21)*. Vol. 35. 5. 2021, pp. 3768–3777.

[GNY19] Stephan Gocht, Jakob Nordström and Amir Yehudayoff. "On Division Versus Saturation in Pseudo-Boolean Solving". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, (IJCAI '19)*. 2019, pp. 1711–1718.

[Goc+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)*. Vol. 12333. Lecture Notes in Computer Science. 2020, pp. 338–357.

[Goc+22] Stephan Gocht, Jakob Nordström, Ruben Martins and Andy Oertel. "Certified CNF Translations for Pseudo-Boolean Solving". In: *Proceedings of the 25nd International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*. 2022.

[Hak85] Armin Haken. "The Intractability of Resolution". In: *Theoretical Computer Science* 39.2-3 (1985), pp. 297–308.

[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)*. 2013, pp. 181–188.

[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)*. Vol. 7898. Lecture Notes in Computer Science. 2013, pp. 345–359.

[HHW15] Marijn J. H. Heule, Warren A. Hunt Jr. and Nathan Wetzler. "Expressing Symmetry Breaking in DRAT Proofs". In: *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*. Vol. 9195. Lecture Notes in Computer Science. 2015, pp. 591–606.

[HJ12] Cheng-Shen Han and Jie-Hong Roland Jiang. "When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way". In: *Proceedings of the International Conference on Computer Aided Verification (CAV '12)*. 2012, pp. 410–426.

[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)*. Vol. 10395. Lecture Notes in Computer Science. 2017, pp. 130–147.

[IPS99] Russell Impagliazzo, Pavel Pudlák and Jiří Sgall. “Lower Bounds for the Polynomial Calculus and the Gröbner Basis Algorithm”. In: *Computational Complexity* 8.2 (1999), pp. 127–144.

[Kar72] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. The IBM Research Symposia Series. 1972, pp. 85–103.

[Kau+22] Daniela Kaufmann, Mathias Fleury, Armin Biere and Manuel Kauers. “Practical algebraic calculus and Nullstellensatz with the checkers Pacheck and Pastèque and Nuss-Checker”. In: *Formal Methods in System Design* (2022).

[KBK20] Daniela Kaufmann, Armin Biere and Manuel Kauers. “From DRUP to PAC and Back”. In: *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '20)*. 2020, pp. 654–657.

[KFB20] Daniela Kaufmann, Mathias Fleury and Armin Biere. “The Proof Checkers Pacheck and Pastèque for the Practical Algebraic Calculus”. In: *Proceedings of Formal Methods in Computer Aided Design, FMCAD 2020*. 2020, pp. 264–269.

[LMS17] Kevin Leyton-Brown, Paul Milgrom and Ilya Segal. “Economics and Computer Science of a Radio Spectrum Reallocation”. In: *Proceedings of the National Academy of Sciences* 114.28 (2017), pp. 7202–7209.

[McC+11] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher and Pascal Schweitzer. “Certifying Algorithms”. In: *Computer Science Review* 5.2 (2011), pp. 119–161.

[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)*. Vol. 7276. Lecture Notes in Computer Science. 2012, pp. 271–282.

[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)*. Vol. 12150. Lecture Notes in Computer Science. 2020, pp. 316–324.

[PB16] *Pseudo-Boolean Competition 2016*. `https://www.cril.univ-artois.fr/PB16/`. 2016.

[PR16] Tobias Philipp and Adrian Rebola-Pardo. "DRAT Proofs for XOR Reasoning". In: *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*. Vol. 10021. Lecture Notes in Computer Science. 2016, pp. 415–429.

[Raz98] Alexander A. Razborov. "Lower Bounds for the Polynomial Calculus". In: *Computational Complexity* 7.4 (1998), pp. 291–324.

[RBK18] Daniela Ritirc, Armin Biere and Manuel Kauers. "A practical polynomial calculus for arithmetic circuit verification". In: *Proceedings of the 3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2'18)*. 2018, pp. 61–76.

[RS] *RoundingSat*. `https://gitlab.com/MIAOresearch/roundingsat`.

[SAT4j] *Sat4j: The Boolean Satisfaction and Optimization Library in Java*. `http://www.sat4j.org/`.

[SGM20] Mate Soos, Stephan Gocht and Kuldeep S. Meel. "Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling". In: *Proceedings of the International Conference on Computer Aided Verification (CAV '20)*. Vol. 12224. Lecture Notes in Computer Science. 2020, pp. 463–484.

[SKM19] Mate Soos, Raghav Kulkarni and Kuldeep S. Meel. "CrystalBall: Gazing in the Black Box of SAT Solving". In: *Proceedings of the 22th International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*. Vol. 11628. Lecture Notes in Computer Science. 2019, pp. 371–387.

[SNC09] Mate Soos, Karsten Nohl and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*. Vol. 5584. Lecture Notes in Computer Science. 2009.

[Vin+18] Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht and Jakob Nordström. “In between resolution and cutting planes: a study of proof systems for pseudo-Boolean SAT solving”. In: *Proceedings of the 25nd International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*. Springer. 2018, pp. 292–310.

[VS10] Michael Veksler and Ofer Strichman. “A Proof-Producing CSP Solver”. In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*. 2010, pp. 204–209.

[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 Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*. Vol. 8561. Lecture Notes in Computer Science. 2014, pp. 422–429.

Scientific publications

Paper A

Certifying Parity Reasoning Efficiently Using Pseudo-Boolean Proofs

Stephan Gocht,[1,2] **Jakob Nordström,**[2,1]
[1] Lund University, Lund, Sweden
[2] University of Copenhagen, Copenhagen, Denmark
stephan.gocht@cs.lth.se, jn@di.ku.dk

Abstract

The dramatic improvements in combinatorial optimization algorithms over the last decades have had a major impact in artificial intelligence, operations research, and beyond, but the output of current state-of-the-art solvers is often hard to verify and is sometimes wrong. For Boolean satisfiability (SAT) solvers proof logging has been introduced as a way to certify correctness, but the methods used seem hard to generalize to stronger paradigms. What is more, even for enhanced SAT techniques such as parity (XOR) reasoning, cardinality detection, and symmetry handling, it has remained beyond reach to design practically efficient proofs in the standard *DRAT* format. In this work, we show how to instead use pseudo-Boolean inequalities with extension variables to concisely justify XOR reasoning. Our experimental evaluation of a SAT solver integration shows a dramatic decrease in proof logging and verification time compared to existing *DRAT* methods. Since our method is a strict generalization of *DRAT*, and readily lends itself to expressing also 0-1 programming and even constraint programming problems, we hope this work points the way towards a unified approach for efficient machine-verifiable proofs for a rich class of combinatorial optimization paradigms.

1 Introduction

Since around the turn of the millennium, combinatorial optimization has been successfully applied to solve an ever increasing range of problems in e.g., resource allocation, scheduling, logistics, and disaster management (Pardalos, Du, and Graham 2013), and more recent applications in biology, chemistry, and medicine (Archibald et al. 2019) include, e.g., protein analysis and design (Allouche et al. 2014; Mann, Will, and Backofen 2008) and kidney transplants (Manlove and O'Malley 2012). Yet other examples are government auctions generating billions of dollars in revenue (Leyton-Brown, Milgrom, and Segal 2017), as well as allocation of education and work opportunities (Manlove 2016; Manlove, McBride, and Trimble 2017) and matching of adoptive families with children (Delorme et al. 2019).

As more and more such problems are dealt with using combinatorial optimization solvers, an urgent question is whether we can trust that the solutions computed by such algorithms are correct and complete. The answer, unfortunately, is currently a clear "no": State-of-the-art solvers sometimes return "solutions" that do not satisfy the constraints or erroneously claim optimality (Cook et al. 2013; Akgün et al. 2018; Gillard, Schaus, and Deville 2019). This can be fatal for applications such as, e.g., chip design, compiler optimization, and combinatorial auctions, where correctness is absolutely crucial, not to speak about when human lives depend on finding the best solutions.

Conventional software testing has made little progress in addressing this problem, and formal verification techniques cannot handle the level of complexity of modern solvers. Instead, the most successful approach to date has been that of *proof logging* in the Boolean satisfiability (SAT) community, where solvers are required to *certify* (McConnell et al. 2011) their answer by outputting also a simple, machine-verifiable proof that this answer is correct. This does not certify the correctness of the solver itself, but it does mean that if it ever produces an incorrect answer (even if due to hardware errors), then this can be detected. A number of different proof logging formats such as *RUP* (Goldberg and Novikov 2003), *TraceCheck* (Biere 2006), *DRAT* (Heule, Hunt Jr., and Wetzler 2013a,b; Wetzler, Heule, and Hunt Jr. 2014), *GRIT* (Cruz-Filipe, Marques-Silva, and Schneider-Kamp 2017), and *LRAT* (Cruz-Filipe et al. 2017) have been developed, with *DRAT* now established as the standard in the SAT competitions (www.satcompetition.org).

A quite natural, and highly desirable, goal would be to extend these proof logging techniques to stronger paradigms such as pseudo-Boolean (PB) optimization, MaxSAT solving, mixed integer programming, and constraint programming, but such attempts have had limited success. Either the proofs require trusting in powerful and complicated rules (as in, e.g., (Veksler and Strichman 2010)), defeating simplicity and verifiability, or they have to justify such rules by long explanations, leading to an exponential slow-down (see (Gange and Stuckey 2019)). In fact, even for SAT solvers a long-standing problem is that more advanced techniques reasoning with parity constraints (XORs), cardinality constraints, and symmetries have remained out of reach for efficient proof logging. Although in theory there should be no problems—*DRAT* is extremely powerful, and can in principle justify such reasoning and much more with at most a polynomial amount of work (Sinz and Biere 2006;

Heule, Hunt Jr., and Wetzler 2015; Philipp and Rebola-Pardo 2016)—in practice the overhead seems completely prohibitive. Thus, a key challenge on the road to efficient proof logging for more general combinatorial optimization solvers would seem to be to design a method that can capture the full range of techniques used in modern SAT solvers.

Our Contribution

In this work, we present a new, efficient proof logging method for parity reasoning that is—perhaps somewhat surprisingly—based on pseudo-Boolean reasoning with 0-1 linear inequalities. Though such inequalities might seem ill-suited to representing XOR constraints, this can be done elegantly by introducing auxiliary so-called *extension variables* (Dixon, Ginsberg, and Parkes 2004). Using this observation, we strengthen the *VeriPB* tool recently introduced in (Elffers et al. 2020), which can be viewed as a generalization to pseudo-Boolean proofs of *RUP* (Goldberg and Novikov 2003). Borrowing inspiration from (Heule, Kiesl, and Biere 2017; Buss and Thapen 2019), we develop stronger, but still efficient, rules that can handle also extension variables, making *VeriPB*, in effect, into a strict generalization of *DRAT*.

We have implemented our method for representing XOR constraints and performing Gaussian elimination in a library with a simple, clean interface for SAT solvers. As a proof of concept, we have also integrated it in *MiniSat* (Eén and Sörensson 2004), which still serves as the foundation of most state-of-the-art SAT solvers. Our library also provides *DRAT* proof logging for XORs as described in (Philipp and Rebola-Pardo 2016), but with some optimizations, to allow for a comparative evaluation. Our experiments show that the overhead for proof logging, the size of the produced proofs, and the time for verification all go down by orders of magnitude for our method compared to *DRAT*. Furthermore, the fact that PB reasoning forms the basis for solvers like *Sat4j* (Le Berre and Parrain 2010) and *RoundingSat* (Elffers and Nordström 2018) means that our library can also empower such pseudo-Boolean solvers to reason with parities.

Since cardinality constraints are just a special case of PB constraints, it is clear that our method should suffice to justify the cardinality reasoning used in SAT solvers. Symmetry reasoning remains a challenge, but at least our method can subsume anything done by *DRAT*. More excitingly, the original *VeriPB* tool has already been shown to be capable of efficiently justifying a number of constraint programming techniques (Elffers et al. 2020; Gocht, McCreesh, and Nordström 2020; Gocht et al. 2020). Our optimistic interpretation is that pseudo-Boolean reasoning with extension variables shows great potential as a unified method of proof logging for SAT solving, pseudo-Boolean optimization, constraint programming, and maybe even mixed integer programming.

Organization of This Paper After some brief background in Section 2, we introduce the key technical notions needed in Section 3 and show how they can be used to justify parity reasoning in Section 4 with a worked out example in Section 5. We present an experimental evaluation in Section 6 and provide some concluding remarks in Section 7.

2 Preliminaries

Let us start by quickly reviewing the required material on pseudo-Boolean reasoning, referring the reader to, e.g., (Buss and Nordström 2021) for more context. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). The set of all literals is denoted *Lits*. For notational convenience, we define $\overline{\overline{x}} = x$. A *pseudo-Boolean (PB) constraint* C is a 0-1 linear inequality

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

which without loss of generality we 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 integers. We will use equality

$$\sum_i a_i \ell_i = A \tag{2a}$$

as syntactic sugar for the pair of inequalities

$$\sum_i a_i \ell_i \geq A \tag{2b}$$

$$\sum_i -a_i \ell_i \geq -A \tag{2c}$$

(but rewritten in normalized form) and the *negation* $\neg C$ of (1) is (the normalized form of)

$$\sum_i -a_i \ell_i \geq -A + 1 \ . \tag{3}$$

A *pseudo-Boolean formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints. Note that a *clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the constraint $\ell_1 + \cdots + \ell_k \geq 1$, so formulas in *conjunctive normal form (CNF)* are special cases of PB formulas.

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$ and a substitution is a (partial) function from variables to *Lits*$\cup\{0, 1\}$. For an assignment or substitution ρ we will use the convention $\rho(x) = x$ for x not in the domain of ρ, denoted $x \notin \mathit{dom}(\rho)$, and define $\rho(\overline{x}) = 1 - \rho(x)$. We also write $x \mapsto b$ instead of $\rho(x) = b$ for $b \in \mathit{Lits} \cup \{0, 1\}$ when ρ is clear from context or is immaterial. Applying ρ to a constraint C as in (1), denoted $C{\restriction_\rho}$, yields the constraint obtained by substituting values for all assigned variables, shifting constants to the right-hand side, and adjusting the degree appropriately, i.e.,

$$C{\restriction_\rho} \ = \ \sum_i a_i \rho(\ell_i) \geq A \tag{4}$$

with appropriate normalization, 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 (4) has a non-positive degree and is thus trivial). A PB formula is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, the formula is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable.

Cutting planes as defined in (Cook, Coullard, and Turán 1987) 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 added. We can also add *literal axioms* $\ell_i \geq 0$ at any time. Finally, from a constraint in normalized form

$\sum_i a_i \cdot \ell_i \geq A$ 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.

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 not hard to see that any collection of constraints F' derived (iteratively) from F by cutting planes are implied in this sense, and so it holds that F and $F \wedge F'$ are equisatisfiable. A piece of terminology that we will use is that C' is *implied syntactically* by C if C' can be derived from C using only addition of literal axioms.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C\restriction_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals $\ell \mapsto 1$ 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 referred to as a *conflict*, since ρ' *violates* the constraint C in this case, and ρ' is called a *conflicting* assignment.

Using the generalization of (Goldberg and Novikov 2003) in (Elffers et al. 2020), we say that F implies C by *reverse unit propagation (RUP)*, and write $RUP(F, C)$, if $F \wedge \neg C$ unit propagates to conflict under the empty assignment. It is not hard to see that $RUP(F, C)$ implies $F \models C$, but the opposite direction is not necessarily true.

3 Substitution Redundancy

In order to provide proof logging for parity reasoning, we need the ability not only to perform cutting planes reasoning, but also to introduce *fresh variables* not occurring in the formula F under consideration. In particular, we want to be able to use a fresh variable y to encode the *reification* of a constraint $\sum_i a_i \ell_i \geq A$, i.e., that y is true if and only if the constraint is satisfied. We will use the shorthand

$$y \leftrightarrow \textstyle\sum_i a_i \ell_i \geq A \tag{5}$$

for the two constraints

$$A\overline{y} + \textstyle\sum_i a_i \ell_i \geq A \tag{6a}$$

$$\left(-A+1+\textstyle\sum_i a_i\right) \cdot y + \sum_i a_i \bar{\ell}_i \geq -A + 1 + \sum_i a_i \tag{6b}$$

enforcing this condition. By way of a concrete example, the reification of the constraint

$$x_1 + x_2 + x_3 \geq 2 \tag{7}$$

using y is encoded as

$$2\overline{y} + x_1 + x_2 + x_3 \geq 2 \tag{8a}$$

$$2y + \overline{x}_1 + \overline{x}_2 + \overline{x}_3 \geq 2 \tag{8b}$$

in pseudo-Boolean form. Note that introducing such constraints maintains equisatisfiability provided that y does not appear in any other constraint, since depending on whether (7) is satisfied or not we can assign y freely to satisfy (8a) or (8b) as needed.

More generally, it would be convenient to allow the "derivation" of any constraint C from F such that F and $F \wedge C$ are equisatisfiable—in which case we say that C is *redundant with respect to* F—regardless of whether $F \models C$ holds or not. A moment of thought reveals that such a completely generic rule would be too good to be true—for any unsatisfiable formula F we would then be able to "derive" contradiction (say, $0 \geq 1$) in just one step, and this clearly would not be efficiently verifiable. What we need, therefore, is a sufficient criterion for redundancy of pseudo-Boolean constraints that is simple to verify. To this end, we generalize the characterization of redundancy in (Heule, Kiesl, and Biere 2017; Buss and Thapen 2019) from CNF formulas to PB formulas as follows.

Proposition 1 (Substitution redundancy)**.** A PB constraint C is redundant with respect to the formula F if and only if there is a substitution ω, called a *witness*, for which it holds that

$$F \wedge \neg C \models (F \wedge C)\restriction_\omega \ .$$

Proof. ($\Rightarrow$) Suppose C is redundant. If F is unsatisfiable, then for any constraint C' it vacuously holds that $F \models C'$. Hence, any substitution ω fulfils the condition. If F is satisfiable, then $F \wedge C$ must also be satisfiable as C is redundant by assumption. If we choose ω to be a satisfying assignment for $F \wedge C$, the implication in the proposition again vacuously holds since $(F \wedge C)\restriction_\omega$ is fixed to true.

($\Leftarrow$) Suppose now that ω is such that $F \wedge \neg C \models (F \wedge C)\restriction_\omega$. If F is unsatisfiable, then every constraint is redundant and there is nothing to check. Otherwise, let α be a (total) satisfying assignment for F. If α also satisfies C, then clearly the constraint is redundant. Now consider the case that α does not satisfy C. If so, α must satisfy $\neg C$ and hence, by the assumed implication, also $(F \wedge C)\restriction_\omega$. But then the assignment β defined by

$$\beta(x) = \begin{cases} \alpha(x) & \text{if } x \notin \mathit{dom}(\omega), \\ \omega(x) & \text{otherwise,} \end{cases} \tag{9}$$

satisfies both C and F (since $(F \wedge C)\restriction_\beta = ((F \wedge C)\restriction_\omega)\restriction_\alpha$ by construction), so $F \wedge C$ is satisfiable. □

We remark that this proof does not make use of that we are operating with a pseudo-Boolean constraint C—we only need that the negation $\neg C$ is easy to represent in the same formalism. Thus, the argument generalizes to other types of constraints with this property.

Let us return to our example reification of the constraint in (7) and show how this can be derived using substitution redundancy. Let us write C_{8a} for the constraint in (8a) and C_{8b} for (8b), where y is fresh with respect to the current formula F. To show that C_{8b} is substitution redundant with respect to F we choose the witness $\omega = \{ y \mapsto 1 \}$, which clearly satisfies C_{8b}. Since y does not appear in F we have $F\restriction_\omega = F$, and so the implication $F \wedge \neg C_{8b} \models (F \wedge C)\restriction_\omega$ vacuously holds. Showing that C_{8a} is substitution redundant with respect to $F \wedge C_{8b}$ is a bit more interesting. For this we choose $\omega = \{ y \mapsto 0 \}$, which satisfies C_{8a} and again leaves F unchanged. Thus, the only implication for which we need to do some work is $F \wedge C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$. The negation of C_{8a} is

$$-2\overline{y} - x_1 - x_2 - x_3 \geq -1 \ , \tag{10}$$

Algorithm 1 Checking substitution redundancy

1: **procedure** REDUNDANCYCHECK(F, C, ω)
2: $\triangleright$ C, ω are given in the proof log to be verified
3: **if** $RUP(F, C)$ **then return** pass
4: **for** $D \in (F \wedge C)\restriction_\omega$ **do**
5: **if** (**not** ($D \in F$ **or** $\neg C \models D$ syntactically)
6: **and not** $RUP(F \wedge \neg C, D)$) **then**
7: **return** fail
8: **return** pass

or, converted to normalized form,

$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 4 \tag{11}$$

using the rewriting rule $\ell = 1 - \bar{\ell}$. Adding the literal axiom $\bar{y} \geq 0$ twice to $\neg C_{8a}$, and using rewriting again to cancel $y + \bar{y} = 1$, we obtain

$$\bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 2, \tag{12}$$

which is $C_{8b}\restriction_\omega$. Hence, $\neg C_{8a}$ syntactically implies $C_{8b}\restriction_\omega$, and so $F \wedge C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$, completing the proof that C_{8a} is redundant with respect to $F \wedge C_{8b}$.

So far, we have not discussed how the implications are verified. Arbitrary implication checks are as hard as determining satisfiability, and hence a certificate that the implication is correct is necessary for efficient verification. One way of providing a certificate could be to exhibit a cutting planes derivation establishing the validity of the implication, as in the example just presented. A more convenient alternative from a proof logging point of view is to follow the lead of *DRAT* and only allow constraints for which the implication can be verified using unit propagation. We describe our pseudo-Boolean version of this method in Algorithm 1. This algorithm is very similar to what is used for checking *DRAT*, except that our unit propagation is on PB constraints rather than clauses and that we need an extra syntactic check on line 5. To see why this check is necessary, note that only unit propagation would fail to certify the correctness of our example above. Assuming for simplicity that $F = \emptyset$, if we try to verify $C_{8b} \wedge \neg C_{8a} \models C_{8b}\restriction_\omega$ by reverse unit propagation we get the constraints

$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 2 \quad [C_{8b} \text{ in (8b)}] \tag{13a}$$
$$2y + \bar{x}_1 + \bar{x}_2 + \bar{x}_3 \geq 4 \quad [\neg C_{8a} \text{ in (11)}] \tag{13b}$$
$$x_1 + x_2 + x_3 \geq 2 \quad [\neg(C_{8b}\restriction_\omega)] \tag{13c}$$

and although visual inspection shows that this collection of constraints is inconsistent, since it requires a majority of the variables $\{ x_1, x_2, x_3 \}$ to be true and false at the same time, unit propagation is too myopic to see this contradiction and only yields $y \mapsto 1$. Using Algorithm 1, however, allows us to introduce extension variables encoding reifications $y \leftrightarrow C$, which is straightforward to prove by carrying out the same argument as in our example above but for (6a) and (6b) instead of (8a) and (8b).

Proposition 2. Let F be a PB formula and C be a PB constraint, and y is a fresh variable that does not appear in F or C. Then the constraints (6a) and (6b) encoding $y \leftrightarrow C$ can be added to F and checked as redundant by Algorithm 1.

4 Proof Logging for XOR Constraints

An *XOR* or *parity constraint*, i.e., an equality modulo 2, over k variables is written as

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \tag{14}$$

where $b \in \{ 0, 1 \}$. Note that we can assume that there is no parity constraint with a negated variable $\bar{x}$, because we can always substitute $\bar{x} = x \oplus 1$.

Systems of XOR constraints can arise in a solver during Gaussian elimination (Soos, Nohl, and Castelluccia 2009; Han and Jiang 2012; Laitinen, Junttila, and Niemelä 2012) or conflict analysis (Laitinen, Junttila, and Niemelä 2012). To provide proof logging for these approaches we need four ingredients:

1. An efficient encoding of XORs to pseudo-Boolean constraints.
2. A way to derive that encoding for a new XOR constraint from the encodings of existing XORs.
3. A method to translate to this efficient encoding from CNF (which is where we will need to go beyond cutting planes by using extension variables).
4. The ability to provide so-called *reason clauses* from the PB encoding that can be used by a SAT solver during conflict analysis.

In the example in Section 5, we will see how these ingredients come together for propagations from parity constraints.

It was observed by (Dixon, Ginsberg, and Parkes 2004) that XOR constraints can be encoded and refuted efficiently in pseudo-Boolean form by rewriting (14) as

$$\textstyle\sum_{i \in [k]} x_i = b + \sum_{i \in [\lfloor k/2 \rfloor]} 2y_i \tag{15}$$

for fresh variables y_i (where we recall that equality (2a) is a shorthand for (2b) and (2c)). Since the variables y_i are otherwise unconstrained, the right-hand side can take any even (odd) value for $b = 0$ ($b = 1$) in the range from 0 to k, which are exactly the values that we want to allow for $\sum_{i \in [k]} x_i$.

In fact, we can generalize this by observing that if we let $\mathcal{B}$ denote any integer linear combination of variables, possibly also with a constant term, then the two inequalities

$$\textstyle\sum_{i \in [k]} x_i \geq b + 2\mathcal{B} \tag{16a}$$
$$\textstyle\sum_{i \in [k]} -x_i \geq -b - 2\mathcal{B} \ , \tag{16b}$$

forming the equality $\sum_{i \in [k]} x_i = b + 2\mathcal{B}$, imply the parity

$$x_1 \oplus x_2 \oplus \cdots \oplus x_k = b \ . \tag{16c}$$

We will make repeated use of this observation below.

XOR Reasoning

Whenever we want to combine two XOR constraints to derive a new XOR constraint, as is done during Gaussian elimination, we only need to add the equalities that imply it. For example, suppose that we want to do the derivation

$$\frac{x_1 \oplus x_2 \oplus x_3 = 1 \qquad x_2 \oplus x_3 \oplus x_4 = 1}{x_1 \oplus x_4 = 0} \tag{17}$$

and assume the XORs are represented in pseudo-Boolean form as $x_1 + x_2 + x_3 = 2y_1 + 1$ and $x_2 + x_3 + x_4 = 2y_2 + 1$. Then adding both equalities together we obtain $x_1 + 2x_2 + 2x_3 + x_4 = 2y_1 + 2y_2 + 2$, which implies the desired XOR by the observation we just made above for (16c).

Reason Generation

Modern SAT solvers built on *conflict-driven clause learning (CDCL)* (Bayardo Jr. and Schrag 1997; Marques-Silva and Sakallah 1999; Moskewicz et al. 2001) operate with clauses. If we want to use XOR constraints to propagate forced variable assignments or derive contradiction, then we need to provide *reason clauses* that justify such derivation steps. We next show how to derive such reason clauses from pseudo-Boolean encodings of XOR constraints.

Suppose we have a parity constraint encoded by inequalities of the form (16a) and (16b), and let ρ be an assignment to the k variables x_i that is inconsistent with (16a) and (16b), because it falsifies the implied XOR (16c). We want to derive a clause that is falsified under ρ.

Let $\mathcal{T}$ be the variables that are set to true under ρ, and $\mathcal{F}$ the variables set to false. Using literal axioms we can derive (the normalized form of) the trivially true constraint

$$\sum_{x\in\mathcal{F}} x + \sum_{x\in\mathcal{T}} -x \geq -|\mathcal{T}| \ , \tag{18}$$

which when added to (16a), yields

$$\sum_{x\in\mathcal{F}} 2x \geq b - |\mathcal{T}| + 2\mathcal{B} \ . \tag{19}$$

Observe that $b - |\mathcal{T}|$ is always odd, as otherwise ρ would not falsify the XOR implied by (16a) and (16b), while everything else is divisible by 2. Hence we can divide by 2, and rounding up will increase the degree. If we now multiply by 2 again, then we get

$$\sum_{x\in\mathcal{F}} 2x \geq b - |\mathcal{T}| + 1 + 2\mathcal{B} \ . \tag{20}$$

We continue by adding (16b) to get

$$\sum_{x\in\mathcal{F}} x - \sum_{x\in\mathcal{T}} x \geq 1 - |\mathcal{T}| \ , \tag{21}$$

which is equivalent to the normalized constraint

$$\sum_{x\in\mathcal{F}} x + \sum_{x\in\mathcal{T}} \overline{x} \geq 1 \ . \tag{22}$$

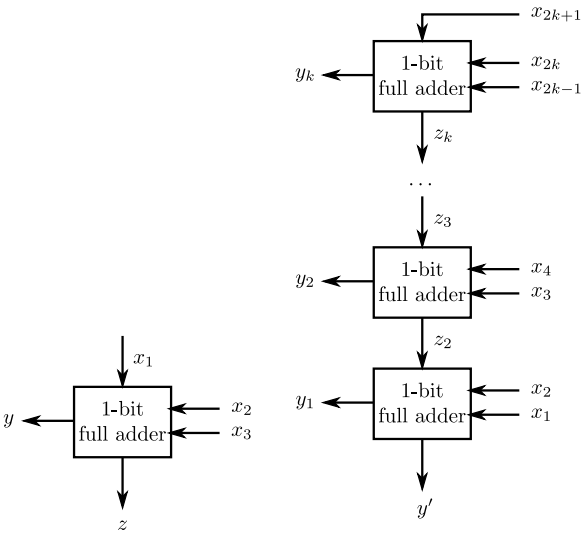
This constraint, which is a disjunctive clause, is falsified under ρ as desired, and is the reason clause that we need to give to the solver to show why the assignment ρ is inconsistent.

Translating to the Pseudo-Boolean XOR Encoding

An XOR as in (14) can be encoded into CNF by having a clause for each of the 2^{k-1} assignments that falsify the constraint. For example, for $k = 3$ and $b = 1$ we get the clauses

$$x_1 + x_2 + x_3 \geq 1 \tag{23a}$$
$$\overline{x}_1 + \overline{x}_2 + x_3 \geq 1 \tag{23b}$$
$$\overline{x}_1 + x_2 + \overline{x}_3 \geq 1 \tag{23c}$$
$$x_1 + \overline{x}_2 + \overline{x}_3 \geq 1 \tag{23d}$$

(in pseudo-Boolean form). Since the number of clauses in this canonical CNF encoding of an XOR constraint scales exponentially with the number of variables, it is only feasible to encode short XORs into CNF in this way. However, it is possible to split up a long XOR into multiple constant-size XORs using auxiliary variables z_i. For example, (14) can be represented as $x_1 \oplus x_2 \oplus z_2 = 0$, $z_2 \oplus x_3 \oplus z_3 = 0$, $\ldots$, $z_{k-2} \oplus x_{k-1} \oplus x_k = b$. If a parity constraint is split up in this way, we only need to re-encode these small parity constraints from CNF into the pseudo-Boolean encoding. Deriving the original long XOR constraint can then be done by XOR reasoning as described earlier.



(a) A 1-bit full adder. (b) Chain of 1-bit full adders.

Figure 1: The output of 1-bit full adders is used to encode new auxiliary Boolean variables, which represent the sum of all Boolean input variables.

The translation to the pseudo-Boolean XOR encoding from CNF is done in two steps. The first step is to derive the constraint

$$\sum_{i\in[k]} x_i = \sum_{i\in[\lfloor k/2 \rfloor]} 2y_i + y', \tag{24}$$

where y_i and y' are all fresh variables. Note that adding (24) to any formula does not change satisfiability because we can always assign the fresh variables so that the equality holds.

Although the constraint is redundant, we cannot use Proposition 1 directly, because we can not construct a witness assignment ω that is independent of the existing x_i variables, and if ω contains any existing variables then the redundancy check in Algorithm 1 may not be strong enough in general. Instead we introduce each fresh variable individually, analogously to what was done in Section 3.

The second step is to brute-force all possible assignments for the variables x_i, which together with the CNF encoding of the XOR allows us to derive $y' = b$, meaning that we have a constraint of the desired form (24). We remark that this brute-force step is polynomial in the number of clauses of the CNF encoding. We now describe this process in detail.

Step 1a. To derive (24) we will construct a chain of 1-bit full adders (see Figure 1b). But let us start first by showing how the encoding of a single 1-bit full adder can be derived. A 1-bit full adder (Figure 1a) computes the sum of three variables x_1 to x_3 and returns the result as a binary number. This can be encoded using the pseudo-Boolean equality

$$2y + z = x_1 + x_2 + x_3. \tag{25}$$

To obtain (25) we start by deriving the reifications

$$y \leftrightarrow x_1 + x_2 + x_3 \geq 2 \tag{26a}$$
$$z \leftrightarrow x_1 + x_2 + x_3 - 2y \geq 1, \tag{26b}$$

for fresh variables y and z using substitution redundancy as described in Proposition 2. Writing the constraints in normalized form yields

$$x_1 + x_2 + x_3 + 2\overline{y} \geq 2 \tag{27a}$$
$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y \geq 2 \tag{27b}$$
$$x_1 + x_2 + x_3 + 2\overline{y} + 3\overline{z} \geq 3 \tag{27c}$$
$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y + 3z \geq 3. \tag{27d}$$

To derive the less-than-or-equal part of (25), which in normalized form is

$$x_1 + x_2 + x_3 + 2\overline{y} + \overline{z} \geq 3 \ , \tag{28}$$

we add Equation (27c) and 2 times Equation (27a) followed by division by 3. In a similar fashion, to derive the greater-than-or-equal part of (25), which in normalized form is

$$\overline{x}_1 + \overline{x}_2 + \overline{x}_3 + 2y + z \geq 3 \ , \tag{29}$$

we add Equation (27d) and 2 times Equation (27b) followed by division by 3.

Step 1b. To derive Equation (24) we use a chain of 1-bit full adders (Figure 1b). The x_i variables are used as input and x_{2k+1}, which is used in the topmost adder, will only occur if the number of variables is odd, otherwise the topmost adder will only have x_{2k} and x_{2k-1} as input. The variables y_i, y' encode the sum of the input variables as required for Equation (24). The z_i variables are intermediate parity bits. For the topmost adder in Figure 1b we thus have

$$2y_k + z_k = x_{2k+1} + x_{2k} + x_{2k-1} \ , \tag{30}$$

for the intermediate adders we have, for $i \in \{\, 2, \ldots, k-1 \,\}$,

$$2y_i + z_i = z_{i+1} + x_{2i} + x_{2i-1} \ , \tag{31}$$

and for the bottom adder we have

$$2y_1 + y' = z_2 + x_2 + x_1 \ . \tag{32}$$

By adding the encoding of all 1-bit adders, i.e., Equations (30) to (32), we obtain

$$\sum_{i=1}^{k} 2y_i + y' + \sum_{i=2}^{k} z_i = \sum_{i=2}^{k} z_i + \sum_{i=1}^{2k+1} x_i \ . \tag{33}$$

Note that $\sum_{i=2}^{k} z_i$ appears on both sides of the equation and hence Equation (33) is equivalent to Equation (24). Indeed, we do not need to remove $\sum_{i=2}^{k} z_i$ explicitly during proof logging because it will disappear automatically due to constraint normalization.

Step 2. The final step is to obtain the value of y' by brute-forcing all values of x_i. Consider an assignment ρ with $\rho(y') = 1 - b$ that additionally assigns all variables x_i and no other variables. Note that there are 2^k such assignments. Either ρ falsifies one of the clauses that encode the XOR, or else $\sum_{i \in [k]} \rho(x_i) \bmod 2 = b \neq \rho(y')$, meaning that Equation (33) is falsified so that we can derive a clause falsified under ρ as described above in our discussion of reason generation. Combining these 2^k clauses together we can obtain a clause that forces y' to take value b, which we can add to Equation (33) to replace y' with a constant value. This concludes the derivation of the pseudo-Boolean encoding (15) of the XOR constraint from a CNF encoding.

5 A Worked-Out Proof Logging Example

Consider the two parity constraints $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_2 \oplus x_3 \oplus x_4 = 1$, which are encoded as clauses by writing

```
* #variable= 4 #constraint= 8
+1 ~x1 +1  x2 +1  x3 >= 1 ;
+1  x1 +1 ~x2 +1  x3 >= 1 ;
+1  x1 +1  x2 +1 ~x3 >= 1 ;
+1 ~x1 +1 ~x2 +1 ~x3 >= 1 ;
+1  x2 +1  x3 +1  x4 >= 1 ;
+1  x2 +1 ~x3 +1 ~x4 >= 1 ;
+1 ~x2 +1  x3 +1 ~x4 >= 1 ;
+1 ~x2 +1 ~x3 +1  x4 >= 1 ;
```

using the standard OPB file format[1]. The solver reads this formula and runs an algorithm to detect clausal encodings of parities. Once a parity is detected, a proof is generated that translates the parity from the clausal encoding into the PB encoding. For this translation it is necessary to introduce fresh variables via substitution redundancy. The proof log contains a line of the form

```
red [constraint C] ; [assignment omega]
```

where `red` identifies the line as a substitution redundancy step, followed by the constraint C to be added and the witness substitution ω, where each variable to be substituted is listed followed by its substitution. A variable and its substitution can optionally be separated by '->'.

The translation from clausal to PB encoding starts with the reification $y_1 \leftrightarrow x_1 + x_2 + x_3 \geq 2$ for the fresh variable y_1. In the proof format this is done by the two lines

```
red 1  x1 1  x2 1  x3 2 ~y1 >= 2 ; y1 -> 0
red 1 ~x1 1 ~x2 1 ~x3 2  y1 >= 2 ; y1 -> 1
```

which can be checked using Algorithm 1 as discussed in Section 3. After the check passes, the verifier adds these two new constraints to the database and assigns them ids 9 and 10 (the ids 1 to 8 are used for the constraints from the input formula). Similarly, we can do another reification $y_2 \leftrightarrow x_1 + x_2 + x_3 - 2y_1 \geq 1$ using the proof lines

```
red 1  x1 1 x2 1 x3 2 ~y1 3 ~y2 >= 3 ; y2 0
red 1 ~x1 1 ~x2 1 ~x3 2 y1 3 y2 >= 3 ; y2 1
```

The variables y_1, y_2 now correspond to the output bits of a single full adder. Because the parity is only over three variables, we do not need to derive a chain of multiple full adders. Note that the constraints arising from reification do not need to be added to the database of the solver—they are only used for the proof log—but the solver should stay clear of using the variables y_1, y_2 for other purposes.

The next step is to combine the constraints we just derived (ids 9 to 12) via a sequence of cutting planes steps, which are

[1] http://www.cril.univ-artois.fr/PB16/format.pdf

written down in reverse polish notation (using the p-rule in *VeriPB*), also known as postfix notation

```
p 11 9 2 * + 3 d
p 12 10 2 * + 3 d
```

The first line starts with the constraint with id 11 and adds two times the constraint with id 9 and then divides by 3 and rounds up. The same operations are done in the second line but with the constraints with ids 12 and 10. The two lines derive the constraints

$$\text{(ID: 13)} \quad x_1 + x_2 + x_3 + 2\bar{y}_1 + \bar{y}_2 \geq 3 \tag{34a}$$

$$\text{(ID: 14)} \quad \bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y_1 + y_2 \geq 3 \ , \tag{34b}$$

which correspond to (24). The constraints in (34a) and (34b) do not correspond to a parity constraint yet, as they can always be satisfied by setting the fresh variables y_1, y_2 to the right value. To get a proper PB encoding of the first parity we need to fix the value of y_2, which can be done by generating a brute-force proof via p-rules that derives $\bar{y}_2 \geq 1$, which gets id 15 and can be used to remove y_2 from (34b). To remove y_2 from (34a) we can simply use the literal axiom $y_2 \geq 0$ which is obtained by writing the literal `y2` in the p-rule. Both steps together can be written as

```
p 13 y2 +
p 14 15 +
```

deriving the inequalities encoding the first parity, namely

$$\text{(ID: 16)} \quad x_1 + x_2 + x_3 + 2\bar{y}_1 \geq 2 \tag{35a}$$

$$\text{(ID: 17)} \quad \bar{x}_1 + \bar{x}_2 + \bar{x}_3 + 2y_1 \geq 3 \ , \tag{35b}$$

which correspond to (15). Analogously, we can derive the pseudo-Boolean encoding for the second parity

$$\text{(ID: 25)} \quad x_2 + x_3 + x_4 + 2\bar{y}_3 \geq 3 \tag{36a}$$

$$\text{(ID: 26)} \quad \bar{x}_2 + \bar{x}_3 + \bar{x}_4 + 2y_3 \geq 4 \ . \tag{36b}$$

This concludes the proof logging done after detecting parities. Note that the detection of parities and the proof generation for translating from clausal to PB encoding is only done once in our implementation at the start of the solver.

Let us now assume that the solver decides $x_1 = 0$. Note that adding the two parities of the formula yields $x_1 \oplus x_4 = 1$, and hence x_4 should propagate to 1, which is detected by the XOR propagator via Gaussian elimination. The proof for this step is to perform the same addition of the parities, but on their PB representations

```
p 16 25 +
p 17 26 +
```

which yields new constraints

$$\text{(ID: 27)} \quad x_1 + 2x_2 + 2x_3 + x_4 + 2\bar{y}_1 + 2\bar{y}_3 \geq 5 \tag{37a}$$

$$\text{(ID: 28)} \quad \bar{x}_1 + 2\bar{x}_2 + 2\bar{x}_3 + \bar{x}_4 + 2y_1 + 2y_3 \geq 7. \tag{37b}$$

These two constraints imply the parity $x_1 \oplus x_4 = 1$ by the observation made in connection with (16a)–(16c).

The reason clause $x_1 + x_4 \geq 1$ provided by the XOR propagator must be derived in the proof. The assignment falsifying the reason clause is $\rho = \{ x_1 \mapsto 0, x_4 \mapsto 0 \}$. Following the approach in (18)–(22), we derive $x_1 + x_4 \geq 0$ and combine it with the constraints representing the parity

Instance	*MiniSat* + XOR (*PBP*)	(*DRAT*)	*PR2DRAT*
Urquhart-s5-b1	76.8	3033.1	3878.4
Urquhart-s5-b2	79.8	2844.4	3575.2
Urquhart-s5-b3	116.9	7584.0	7521.0
Urquhart-s5-b4	94.7	5058.6	5271.5

Table 1: Proof sizes (KiB) for previous Tseitin formulas.

```
p 27 x1 x4 + + 2 d 2 * 28 +
```

which derives the constraint $x_1 + x_4 \geq 1$ as desired. The solver can continue using this clause in the same way as any other clause in its database. These steps for XOR reasoning and reason generation are repeated for every propagation.

6 Implementation and Evaluation

As just illustrated in Section 5, we have added a rule to *VeriPB*[2] and its pseudo-Boolean proof format (*PBP*) to support redundancy checks as described in Algorithm 1, and have implemented our proof logging approach for XOR reasoning in a library together with an XOR engine using Gaussian elimination $\bmod 2$ to detect XOR propagations.[3] We integrated this library into *MiniSat* to call the XOR propagation method every time propagation reached fix point. If the library detects a propagation or conflict a callback is used to notify *MiniSat*, but the reason clause is only generated when needed in conflict analysis. This *lazy reason generation* technique (Soos, Gocht, and Meel 2020) is crucial, since it avoids generating proofs for reasons that are not used. For comparison, our library also provides *DRAT* proof logging for XORs as described in (Philipp and Rebola-Pardo 2016).

Importantly, our goal was not to investigate whether XOR reasoning is useful—this is already known—but to provide efficient proof logging for such reasoning. Therefore, we focused on SAT competition benchmarks from the last 5 years that could be solved by *MiniSat* with our XOR propagator but not by *Kissat*, the winner of the 2020 SAT competition. There were 39 such instances, and they could be solved in 0.03 seconds on average by *MiniSat* with the XOR propagator. With our new proof logging the average running time increased to 0.05 seconds and unsatisfiability could be verified in 1.71 seconds on average. For *DRAT* proof logging, on the other hand, the average solving time jumped to 7.72 seconds and verification took 3291 seconds on average.

In order to get systematic measurements for the performance of our new proof logging technique, we ran experiments on so-called Tseitin formulas, including some that have been studied before in the context of proof logging. Tseitin formulas consist of a large inconsistent set of parity constraints, and can thus be viewed as a worst case for XOR reasoning. To the best of our knowledge, the shortest DRAT proofs[4] for these formulas obtained so far are based on hand-crafted so-called *propagation redundancy*

[2] https://gitlab.com/miao%5Fresearch/veripb

[3] https://gitlab.com/miao%5Fresearch/xorengine

[4] The proofs and instances can be found at https://github.com/marijnheule/drat2er-proofs

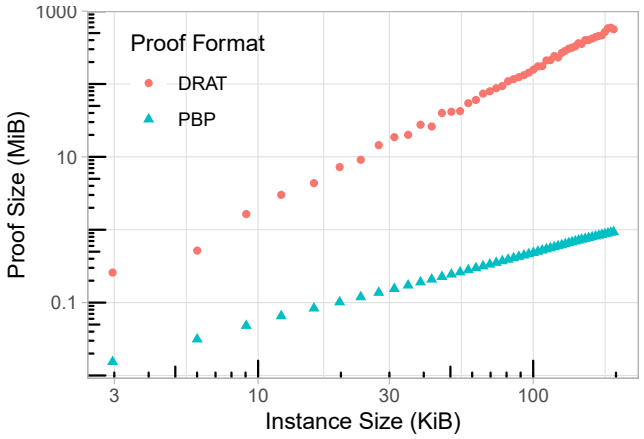


Figure 2: Proof sizes for Tseitin formulas.

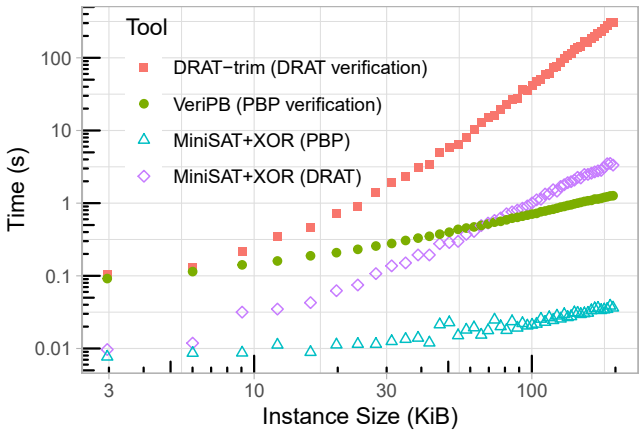


Figure 3: Solving and verification time for Tseitin formulas.

(PR) proofs which have been translated to *DRAT* using the tool *PR2DRAT* (Kiesl, Rebola-Pardo, and Heule 2018). Table 1 shows the disk space required for the proofs of Tseitin formulas in (Kiesl, Rebola-Pardo, and Heule 2018). The pseudo-Boolean proofs obtained by *MiniSat* with the XOR propagator are dramatically smaller than the DRAT proofs it produces, and the size of our DRAT proofs are similar to that of the best previously known DRAT proofs.

To get a sense of the asymptotic behaviour of the proof logging we generated 50 new, larger Tseitin formulas with up to 500 XORs and up to 1250 variables. In Figure 2 we compare the proof size of the proofs as generated. Notice that both proof logging approaches result in a straight line in the log-log plot, which is a strong indication that both approaches are polynomial. Studying the slopes of the lines yields the estimates that *DRAT* produces quadratic-size proofs while the proof size of the pseudo-Boolean proof is linear in the size of the formula. In Figure 3 we compare the running time (system time + user time) of solving and producing the proof, as well as time spend for verification. It is clear that the larger proof size required for *DRAT* proofs does not only increase verification time, but also causes a clearly increased time overhead during solving. All running times were measured on an Intel® Core™ i3-7100U CPU @ 2.40GHz×2 with a memory limit of 8GB, disk write speed of 154 MB/s and read speed of 518 MB/s. The used tools, benchmarks, data and evaluation scripts are available at https://doi.org/10.5281/zenodo.4569840.

7 Conclusion

In this work, we present an efficient method for proof logging parity reasoning in conflict-driven clause learning (CDCL) solvers, which has been a long-standing challenge in SAT solving. Our method circumvents the prohibitive overhead of current *DRAT*-based methods by instead using pseudo-Boolean inequalities with extension variables. An experimental evaluation shows that this makes the proof logging overhead, the size of the proof, and the time required for verification all go down by an order of magnitude or more compared to *DRAT*. While there is certainly ample room for further improvements, our first proof-of-concept implementation already shows the power of this approach.

It is clear that the same method can also be used to solve another task that has remained very challenging for *DRAT*, namely efficient proof logging for cardinality detection and reasoning. We have not investigated this in this paper, since this is mostly an engineering issue rather than a research problem, given the methods that have already been developed in (Biere et al. 2014; Elffers and Nordström 2020).

Dealing with symmetries appears to be much more challenging, but our method can do at least as well as (Heule, Hunt Jr., and Wetzler 2015) since it is a strict generalization of *DRAT*. We would therefore propose that the current extension of the *VeriPB* method in (Elffers et al. 2020) should be an allowed proof logging format in the SAT competitions. This would make it possible for solvers making use of these advanced techniques to take part in the main track of the SAT competitions, where proof logging is mandatory.

However, we believe that the potential benefit of PB proof logging with extension variables goes well beyond the SAT competitions. The original *VeriPB* method is capable of efficient justification of important constraint programming techniques (Elffers et al. 2020), and can also provide proof logging for a wide range of graph problem solvers (Gocht, McCreesh, and Nordström 2020; Gocht et al. 2020). The pseudo-Boolean rules for reasoning with 0-1 linear constraints provide a simple yet very expressive formalism, and it does not seem out of the question to hope that they could be extended to deal with mixed integer programming (MIP). Thus, we believe that the ultimate goal of this line of research should be to design a unified proof logging approach for as wide as possible a range of combinatorial optimization paradigms. In addition to furnishing efficient machine-verifiable proofs of correctness, proof logging could also serve as a valuable tool for debugging and empirical performance analysis during solver development. Furthermore, the proofs produced could in principle provide auditability by third parties using independently developed software, and/or be a stepping stone towards explainability by showing, e.g., why certain solutions are optimal.

Acknowledgments

We want to thank Mate Soos and Kuldeep Meel for helpful discussions on how to implement Gaussian elimination modulo 2. The authors 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.

References

Akgün, Ö.; Gent, I. P.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2018. Metamorphic Testing of Constraint Solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP '18)*, volume 11008 of *LNCS*, 727–736. Springer.

Allouche, D.; André, I.; Barbe, S.; Davies, J.; Givry, S. d.; Katsirelos, G.; O'Sullivan, B.; Prestwich, S.; Schiex, T.; and Traoré, S. 2014. Computational Protein Design as an Optimization Problem. *Artificial Intelligence* 212(1): 59–79.

Archibald, B.; Dunlop, F.; Hoffmann, R.; McCreesh, C.; Prosser, P.; and Trimble, J. 2019. Sequential and Parallel Solution-Biased Search for Subgraph Algorithms. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '19)*, volume 11494 of *LNCS*, 20–38. Springer.

Bayardo Jr., R. J.; and Schrag, R. 1997. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, 203–208.

Biere, A. 2006. TraceCheck. http://fmv.jku.at/tracecheck/. Accessed on 2021–03–19.

Biere, A.; Le Berre, D.; Lonca, E.; and Manthey, N. 2014. Detecting Cardinality Constraints in CNF. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *LNCS*, 285–301. Springer.

Buss, S. R.; and Nordström, J. 2021. Proof Complexity and SAT Solving. In Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 7, 233–350. IOS Press, 2nd edition.

Buss, S. R.; and Thapen, N. 2019. DRAT Proofs, Propagation Redundancy, and Extended Resolution. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT '19)*, volume 11628 of *LNCS*, 71–89. Springer.

Cook, W.; Coullard, C. R.; and Turán, G. 1987. On the Complexity of Cutting-Plane Proofs. *Discrete Applied Mathematics* 18(1): 25–38.

Cook, W.; Koch, T.; Steffy, D. E.; and Wolter, K. 2013. A Hybrid Branch-and-Bound Approach for Exact Rational Mixed-Integer Programming. *Mathematical Programming Computation* 5(3): 305–344.

Cruz-Filipe, L.; Heule, M. J. H.; Hunt, W. A.; Kaufmann, M.; and Schneider-Kamp, P. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *LNCS*, 220–236. Springer.

Cruz-Filipe, L.; Marques-Silva, J.; and Schneider-Kamp, P. 2017. 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*, 118–135. Springer.

Delorme, M.; García, S.; Gondzioa, J.; Kalcsics, J.; Manlove, D.; and Pettersson, W. 2019. Mathematical Models for Stable Matching Problems with Ties and Incomplete Lists. *European Journal of Operational Research* 277(2): 426–441.

Dixon, H. E.; Ginsberg, M. L.; and Parkes, A. J. 2004. Generalizing Boolean Satisfiability I: Background and Survey of Existing Work. *Journal of Artificial Intelligence Research* 21: 193–243.

Eén, N.; and Sörensson, N. 2004. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *LNCS*, 502–518. Springer.

Elffers, J.; Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Justifying All Differences Using Pseudo-Boolean Reasoning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, 1486–1494.

Elffers, J.; and Nordström, J. 2018. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, 1291–1299.

Elffers, J.; and Nordström, J. 2020. A Cardinal Improvement to Pseudo-Boolean Solving. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI '20)*, 1495–1503.

Gange, G.; and Stuckey, P. 2019. Certifying Optimality in Constraint Programming. Presentation at KTH Royal Institute of Technology. Slides available at https://www.kth.se/polopoly_fs/1.879851.1550484700!/CertifiedCP.pdf.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. SolverCheck: Declarative Testing of Constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP '19)*, volume 11802 of *LNCS*, 565–582. Springer.

Gocht, S.; McBride, R.; McCreesh, C.; Nordström, J.; Prosser, P.; and Trimble, J. 2020. 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 *LNCS*, 338–357. Springer.

Gocht, S.; McCreesh, C.; and Nordström, J. 2020. Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI '20)*, 1134–1140.

Goldberg, E.; and Novikov, Y. 2003. Verification of Proofs of Unsatisfiability for CNF Formulas. In *Proceedings of*

the Conference on Design, Automation and Test in Europe (DATE '03), 886–891.

Han, C.; and Jiang, J. R. 2012. When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way. In *Proceedings of the 24th International Conference on Computer Aided Verification, (CAV '12)*, volume 7358 of *LNCS*, 410–426. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013a. Trimming While Checking Clausal Proofs. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD '13)*, 181–188.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2013b. Verifying Refutations with Extended Resolution. In *Proceedings of the 24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *LNCS*, 345–359. Springer.

Heule, M. J. H.; Hunt Jr., W. A.; and Wetzler, N. 2015. Expressing Symmetry Breaking in DRAT Proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *LNCS*, 591–606. Springer.

Heule, M. J. H.; Kiesl, B.; and Biere, A. 2017. Short Proofs Without New Variables. In *Proceedings of the 26th International Conference on Automated Deduction (CADE-26)*, volume 10395 of *LNCS*, 130–147. Springer.

Kiesl, B.; Rebola-Pardo, A.; and Heule, M. J. H. 2018. Extended Resolution Simulates DRAT. In *Proceedings of the 9th International Joint Conference on Automated Reasoning (IJCAR '18)*, volume 10900 of *LNCS*, 516–531. Springer.

Laitinen, T.; Junttila, T.; and Niemelä, I. 2012. Conflict-Driven XOR-Clause Learning. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT '12)*, volume 7317 of *LNCS*, 383–396. Springer.

Laitinen, T.; Junttila, T.; and Niemelä, I. 2012. Extending Clause Learning SAT Solvers with Complete Parity Reasoning. In *Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence (ICTAI '12)*, 65–72.

Le Berre, D.; and Parrain, A. 2010. The Sat4j Library, Release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7: 59–64.

Leyton-Brown, K.; Milgrom, P.; and Segal, I. 2017. Economics and Computer Science of a Radio Spectrum Reallocation. *Proceedings of the National Academy of Sciences* 114(28): 7202–7209.

Manlove, D. F. 2016. Hospitals/Residents Problem. In Kao, M.-Y., ed., *Encyclopedia of Algorithms*, 926–930. Springer New York.

Manlove, D. F.; McBride, I.; and Trimble, J. 2017. "Almost-stable" Matchings in the Hospitals / Residents Problem with Couples. *Constraints* 22(1): 50–72.

Manlove, D. F.; and O'Malley, G. 2012. 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 *LNCS*, 271–282. Springer.

Mann, M.; Will, S.; and Backofen, R. 2008. CPSP-tools – Exact and Complete Algorithms for High-Throughput 3D Lattice Protein Studies. *BMC Bioinformatics* 9: 230:1–230:8.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers* 48(5): 506–521. Preliminary version in *ICCAD '96*.

McConnell, R. M.; Mehlhorn, K.; Näher, S.; and Schweitzer, P. 2011. Certifying Algorithms. *Computer Science Review* 5(2): 119–161.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 530–535.

Pardalos, P. M.; Du, D.-Z.; and Graham, R. L., eds. 2013. *Handbook of Combinatorial Optimization*. Springer, 2nd edition.

Philipp, T.; and Rebola-Pardo, A. 2016. DRAT Proofs for XOR Reasoning. In *Proceedings of the 15th European Conference on Logics in Artificial Intelligence (JELIA '16)*, volume 10021 of *LNCS*, 415–429. Springer.

Sinz, C.; and Biere, A. 2006. Extended Resolution Proofs for Conjoining BDDs. In *Proceedings of the 1st International Computer Science Symposium in Russia (CSR '06)*, volume 3967 of *LNCS*, 600–611. Springer.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling. In *Proceedings of the 32nd International Conference on Computer Aided Verification, (CAV '20)*, volume 12224 of *LNCS*, 463–484. Springer.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT Solvers to Cryptographic Problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *LNCS*, 244–257. Springer.

Veksler, M.; and Strichman, O. 2010. A Proof-Producing CSP Solver. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI '10)*, 204–209.

Wetzler, N.; Heule, M. J. H.; and Hunt Jr., W. A. 2014. 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 *LNCS*, 422–429. Springer.

Paper B

Certified Symmetry and Dominance Breaking for Combinatorial Optimisation

Bart Bogaerts[1], Stephan Gocht[2,3], Ciaran McCreesh[4], and Jakob Nordström[3,2]

[1]Vrije Universiteit Brussel, Brussels, Belgium
[2]Lund University, Lund, Sweden
[3]University of Copenhagen, Copenhagen, Denmark
[4]University of Glasgow, Glasgow, UK

Abstract

Symmetry and dominance breaking can be crucial for solving hard combinatorial search and optimisation problems, but the correctness of these techniques sometimes relies on subtle arguments. For this reason, it is desirable to produce efficient, machine-verifiable certificates that solutions have been computed correctly. Building on the cutting planes proof system, we develop a certification method for optimisation problems in which symmetry and dominance breaking are easily expressible. Our experimental evaluation demonstrates that we can efficiently verify fully general symmetry breaking in Boolean satisfiability (SAT) solving, thus providing, for the first time, a unified method to certify a range of advanced SAT techniques that also includes XOR and cardinality reasoning. In addition, we apply our method to maximum clique solving and constraint programming as a proof of concept that the approach applies to a wider range of combinatorial problems.

1 Introduction

Symmetries pose a challenge when solving hard combinatorial problems. For example, consider the Crystal Maze puzzle[1] shown in Figure 1, which is often used in introductory constraint modelling courses. A human modeller might notice that the puzzle is the same under a vertical mirror symmetry, and could introduce the constraint $A < G$ to eliminate this. Or, they may notice a horizontal mirror symmetry, which could be broken with $A < B$. Alternatively, they might spot that the *values* are symmetrical, and that we can interchange 1 and 8, 2 and 7, and so on; this can be eliminated by saying that $A \leq 4$. In each case a constraint is being added that preserves satisfiability overall, but that restricts a solver to finding (ideally) just one witness from each equivalence class of solutions—the hope is that this will improve solver performance. However, although we may be reasonably sure that any of these three constraints is correct individually, are combinations of these constraints valid simultaneously? What if we had said $F < C$ instead of $A < B$? And what if we could use numbers more than once? Getting symmetry elimination constraints right can be error-prone even for experienced modellers, and when dealing with larger problems with many constraints and interacting symmetries it can be hard to tell whether an instance is genuinely unsatisfiable, or was made so by an incorrect symmetry constraint.

[1]https://theconversation.com/what-problems-will-ai-solve-in-future-an-old-british-gameshow-can-help-explain-49080

Despite these difficulties, symmetry elimination using both manual and automatic techniques has been key to many successes across modern combinatorial optimisation paradigms such as constraint programming (CP) [Garcia de la Banda et al., 2014], Boolean satisfiability (SAT) [Biere et al., 2021], and mixed-integer programming (MIP) [Achterberg and Wunderling, 2013]. As these optimisation technologies are increasingly being used for high-value and life-affecting decision-making processes, it becomes vital that we can trust their outputs—and unfortunately, current solvers do not always produce the correct answer [Brummayer et al., 2010, Cook et al., 2013, Akgün et al., 2018, Gillard et al., 2019]. The most promising way to address this problem appears to be to use *certification*, or *proof logging*, where a solver must produce an efficiently machine-verifiable certificate that the solution given is correct [Alkassar et al., 2011, McConnell et al., 2011]. This approach has been successfully used in the SAT community, with numerous proof logging formats such as *RUP* [Goldberg and Novikov, 2003], *TraceCheck* [Biere, 2006], *DRAT* [Heule et al., 2013a,b, Wetzler et al., 2014], *GRIT* [Cruz-Filipe et al., 2017b], and *LRAT* [Cruz-Filipe et al., 2017a]. However, currently used methods work only for decision problems, and do not support the full range of SAT solving techniques, let alone CP and MIP solving. As a case in point, there is no efficient proof logging for symmetry breaking, except for limited cases with small symmetries which can interact only in simple ways [Heule et al., 2015]. Tchinda and Djamégni [2020] recently proposed a proof logging method *DSRUP* for symmetric learning of variants of derived clauses, but this format does not support symmetry breaking (in the sense just discussed) and is also inherently unable to support pre- and inprocessing techniques, which are crucial in state-of-the-art SAT solvers.

In this work, we develop a proof logging method for *optimisation* problems, where we are given a formula F and an objective function f, that can deal with *dominance*, a generalization of symmetry. Dominance breaking starts from the observation that we can strengthen F by imposing a constraint C if every solution of F that does not satisfy C is dominated by another solution of F. This technique is used in many fields of combinatorial optimisation [Walsh, 2006, 2012, Gent et al., 2006, McCreesh and Prosser, 2016, Jouglet and Carlier, 2011, Gebser et al., 2011, Bulhões et al., 2018, Hoogeboom et al., 2020, Baptiste and Pape, 1997, Demeulemeester and Herroelen, 2002]. The core idea of our method is to present an *explicit construction* of the dominating solution, so that a verifier can check that this construction strictly improves the objective value and preserves satisfaction of F. This constructed solution might itself be dominated, and hence not satisfy C, but since the objective value decreases with every application, the process must eventually terminate. Importantly, verification does not require construction of an assignment satisfying C, and can be performed efficiently even when multiple constraints are to be added; this resolves a practical issue with earlier approaches like [Heule et al., 2015], which struggle with large or overlapping symmetries. Following preliminaries in Section 2, we describe this method in full detail in Section 3.

We have developed a proof format and verifier on top of *VeriPB* [Elffers et al., 2020, Gocht and Nordström, 2021, Gocht et al., 2020b,a]. The pseudo-Boolean constraints and cutting planes proof system [Cook et al., 1987] used by *VeriPB* are convenient to express and reason with dominance inequalities, and more-

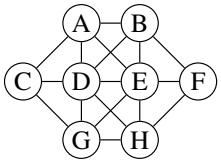


Figure 1: The Crystal Maze puzzle. Place numbers 1 to 8 in the circles, with every circle getting a different number, so that adjacent circles do not have consecutive numbers.

over also make it possible to certify XOR and cardinality reasoning [Gocht and Nordström, 2021], two other advanced techniques which previous SAT proof logging methods have not been able to support efficiently. In Section 4, we demonstrate that our new verifier can efficiently check *automated static symmetry breaking* in SAT, *manual static symmetry breaking* in CP, and *automated dynamic dominance handling* in maximum clique solving. While the latter two applications are proofs of concept, for static symmetry breaking we show in full generality, and for the first time, that proof logging is practical by running experiments on SAT competition benchmarks. We conclude the paper with some brief remarks in Section 5.

2 Preliminaries

Let us briefly review some standard material, referring the reader to, e.g., Buss and Nordström [2021] for more details. A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). A *pseudo-Boolean (PB) constraint* is a 0–1 linear inequality

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

where a_i and A are integers (and $\doteq$ denotes syntactic equality). We can assume without loss of generality that PB constraints are *normalized*; i.e., that all literals ℓ_i are over distinct variables and that the *coefficients* a_i and the *degree (of falsity)* A are non-negative, but most of the time we will not need this. Instead, we will write PB constraints in more relaxed form as $\sum_i a_i \ell_i \geq A + \sum_j b_j \ell_j$ or $\sum_i a_i \ell_i \leq A + \sum_j b_j \ell_j$ when convenient, or even use equality $\sum_i a_i \ell_i = A$ as syntactic sugar for the pair of inequalities $\sum_i a_i \ell_i \geq A$ and $\sum_i -a_i \ell_i \geq -A$, assuming that all constraints are implicitly normalized if needed. The *negation* $\neg C$ of the constraint C in (1) is

$$\neg C \doteq \textstyle\sum_i - a_i \ell_i \geq -A + 1 \ . \tag{2}$$

A *pseudo-Boolean formula* is a conjunction $F \doteq \bigwedge_j C_j$ of PB constraints, which we can also think of as the set $\bigcup_j \{C_j\}$ of constraints in the formula, choosing whichever viewpoint seems most convenient. Note that a *(disjunctive) clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the PB constraint $\ell_1 + \cdots + \ell_k \geq 1$, so formulas in *conjunctive normal form (CNF)* are special cases of PB formulas.

A *(partial) assignment* is a (partial) function from variables to $\{0, 1\}$; a *substitution* can also map variables to literals. We extend an assignment or substitution ρ from variables to literals in the natural way by respecting the meaning of negation, and for literals ℓ over variables x not in the domain of ρ, denoted $x \notin \mathit{dom}(\rho)$, we use the convention $\rho(\ell) = \ell$. (That is, we can consider all assignments and substitution to be total, but to be the identity outside of their specified domains. Strictly speaking, we also require that all substitutions be defined on the truth constants $\{0, 1\}$ and be the identity on these constants.) We sometimes write $x \mapsto b$ when $\rho(x) = b$, for b a literal or truth value.

We write $\rho \circ \omega$ to denote the composed substitution resulting from applying first ω and then ρ, i.e., $\rho \circ \omega(x) = \rho(\omega(x))$. As an example, for $\omega = \{x_1 \mapsto 0, x_3 \mapsto \overline{x}_4, x_4 \mapsto x_3\}$ and $\rho = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0\}$ we have $\rho \circ \omega = \{x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 1, x_4 \mapsto 0\}$. Applying ω to a constraint C as in (1) yields

$$C{\restriction_\omega} \doteq \textstyle\sum_i a_i \omega(\ell_i) \geq A \ , \tag{3}$$

substituting literals or values as specified by ω. For a formula F we define $F{\restriction_\omega} \doteq \bigwedge_j C_j{\restriction_\omega}$.

Since we will sometimes have to make fairly elaborate use of substitutions, let us discuss some further notational conventions. If F is a formula over variables $\vec{x} = \{x_1, \ldots, x_m\}$, we can write $F(\vec{x})$ when we want to stress the set of variables over which F is defined. For a substitution ω with domain (contained in) $\vec{x}$, the notation $F\big(\vec{x}{\restriction_\omega}\big)$ is understood to be a synonym of $F{\restriction_\omega}$. For the same formula F and $\vec{y} = \{y_1, \ldots, y_m\}$, the

notation $F(\vec{y})$ is syntactic sugar for $F\restriction_\omega$ with ω denoting the substitution (implicitly) defined by $\omega(x_i) = y_i$ for $i = 1, \ldots, n$. Finally, for a formula $G = G(\vec{x}, \vec{y})$ over $\vec{x} \cup \vec{y}$ and substitutions α and β defined on $\vec{z} = \{z_1, \ldots, z_n\}$ (either of which could be the identity), the notation $G(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ should be understood as $G\restriction_\omega$ for ω defined by $\omega(x_i) = \alpha(z_i)$ and $\omega(y_i) = \beta(z_i)$ for $i = 1, \ldots, n$.

The (normalized) constraint C in (1) is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq A$. A PB formula F is satisfied by ρ if all constraints in it are, in which case it is *satisfiable*. If there is no satisfying assignment, F is *unsatisfiable*. Two formulas are *equisatisfiable* if they are both satisfiable or both unsatisfiable. We also consider optimisation problems, where in addition to F we are given an integer linear objective function $f \doteq \sum_i w_i \ell_i$ and the task is to find an assignment that satisfies F and minimizes f. (To deal with maximization problems we can just negate the objective function.)

Cutting planes [Cook et al., 1987] is a method for iteratively deriving constraints C from a pseudo-Boolean formula F. We write $F \vdash C$ for any constraint C derivable as follows. Any *axiom constraint* $C \in F$ is trivially derivable, as is any *literal axiom* $\ell \geq 0$. If $F \vdash C$ and $F \vdash D$, then any positive integer *linear combination* of these constraints is derivable. Finally, from a constraint in normalized form $\sum_i a_i \ell_i \geq A$ 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. For a set of PB constraints F' we write $F \vdash F'$ if $F \vdash C$ for all $C \in F'$.

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 also satisfies C, and write $F \models F'$ if $F \models C'$ for all $C' \in F'$. It is easy to see that if $F \vdash F'$ then $F \models F'$, and so F and $F \wedge F'$ are equisatisfiable. A constraint C is said to *literal-axiom-imply* another constraint C' if C' can be derived from C by addition of literal axioms $\ell \geq 0$.

A constraint C *unit propagates* the literal ℓ under ρ if $C\restriction_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, ρ is extended iteratively by any propagated literals until an assignment ρ' is reached under which no constraint $C \in F$ is propagating, or under which some constraint C would propagate a literal had it not already been assigned to the opposite value. The latter scenario is referred to as a *conflict*, since ρ' *violates* the constraint C in this case, and ρ' is called a *conflicting* assignment. Using the generalization of [Goldberg and Novikov, 2003] in [Elffers et al., 2020], 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. If C is a RUP constraint with respect to F, then it can be proven that there is also a derivation $F \vdash C$. More generally, it can be shown that $F \vdash C$ if and only if $F \wedge \neg C \vdash \bot$, where $\bot$ is a shorthand for the *trivially false constraint* $0 \geq 1$. Therefore, we will extend the notation and write $F \vdash C$ also when C is derivable from F by RUP or by contradiction. It is worth noting here again that, as shown in (2), the negation of any PB constraint can also be expressed syntactically as a PB constraint—this fact will be convenient in what follows.

3 A Proof System for Dominance Breaking

We proceed to develop our formal proof system for verifying dominance breaking, which we have implemented on top of the version of *VeriPB* in [Gocht and Nordström, 2021]. We remark that for applications it is absolutely crucial not only that the proof system be sound, but that all proofs be efficiently machine-verifiable. There are significant challenges involved in making proof logging and verification efficient, but in this section we mostly ignore these aspects of our work and focus on the theoretical underpinnings.

Our foundation is the cutting planes proof system described in Section 2. However, in a proof in our system for (F, f), where f is a linear objective function to be minimized under the pseudo-Boolean formula F (or where $f \doteq 0$ for decision problems), we also allow strengthening F by adding constraints C that are not implied by the formula. Pragmatically, adding C should be in order as long as we keep some optimal

solution, i.e., a satisfying assignment to F that minimizes f, which we will refer to as an *f-minimal solution* of F. We will formalize this idea by allowing the use of an additional pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ that, together with a sequence of variables $\vec{z}$, defines a relation $\alpha \preceq \beta$ to hold between assignments α and β if $\mathscr{O}_{\preceq}(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ evaluates to true. We require (a cutting planes proof) that $\mathscr{O}_{\preceq}$ is such that this defines a preorder, i.e., a reflexive and transitive relation. Adding new constraints C will be valid as long as we guarantee to preserve some f-minimal solution that is also minimal with respect to $\preceq$. In other words, $\preceq$ can be combined with f to define a preorder $\preceq_f$ on assignments by

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\restriction_\alpha \leq f\restriction_\beta \ , \tag{4}$$

and we require that all derivation steps in the proof should preserve some solution that is minimal with respect to $\preceq_f$. The preorder defined by $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ will only become important once we introduce our new *dominance-based strengthening rule* later in this section. For simplicity, up until that point the reader can assume that the pseudo-Boolean formula is $\mathscr{O}_\top \doteq \emptyset$ inducing the trivial preorder relating all assignments, though all proofs presented below work in full generality for the orders that will be introduced later.

A proof for (F, f) in our proof system consists of a sequence of *proof configurations* $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$, where

- $\mathscr{C}$ is a set of pseudo-Boolean *core constraints*;
- $\mathscr{D}$ is another set of pseudo-Boolean *derived constraints*;
- $\mathscr{O}_{\preceq}$ is a PB formula encoding a preorder and $\vec{z}$ a set of literals on which this preorder will be applied; and
- v is the best value found so far for f.

The initial configuration is $(F, \emptyset, \mathscr{O}_\top, \emptyset, \infty)$. The distinction between $\mathscr{C}$ and $\mathscr{D}$ is only relevant when a nontrivial preorder is used; we will elaborate on this when discussing dominance. The intended semantics of f and v is that if $v < \infty$, then there exists a solution α satisfying F such that $f\restriction_\alpha \leq v$, and in this case the proof can make use of the constraint $f \leq v - 1$ in the search for better solutions. As long as the optimal solution has not been found, it should hold that f-minimal solutions of $\mathscr{C} \cup \mathscr{D}$ have the same objective value as f-minimal solutions of F. The precise relation is formalized in the notion of *valid configurations* as defined next.

Definition 1. *A configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is* (F, f)-valid *if the following conditions hold:*

1. *If $v < \infty$, then there is a total assignment ρ satisfying F such that $f\restriction_\rho \leq v$.*
2. *For every $v' < v$, it holds that the sets $F \cup \{f \leq v'\}$ and $\mathscr{C} \cup \{f \leq v'\}$ are equisatisfiable.*
3. *For every total assignment ρ satisfying the constraints $\mathscr{C} \cup \{f \leq v - 1\}$, there exists a total assignment $\rho' \preceq_f \rho$ satisfying $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v - 1\}$.*

We will show that (F, f)-validity is an invariant of our proof system, i.e., that it is preserved by all derivation rules. Note that the two last items together imply that if the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is such that v is not yet the value of an optimal solution, then f-minimal solutions of F and of $\mathscr{C} \cup \mathscr{D}$ have the same objective value, just as desired.

A proof in our proof system ends when the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v^*)$ is such that $\mathscr{C} \cup \mathscr{D}$ contains contradiction $\bot \doteq 0 \geq 1$. In that case, either $v^* = \infty$ and F is unsatisfiable, or v^* is the optimal value (or $v^* = 0$ for a satisfiable decision problem). We state this as a formal theorem (but due to space constraints, proofs of all statements in this section can be found in Appendix B).

Theorem 2. *Let F be a pseudo-Boolean formula and f an objective function. If $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v^*)$ is an (F, f)-valid configuration with $\{0 \geq 1\} \subseteq \mathscr{C} \cup \mathscr{D}$, then*

- *F is unsatisfiable if and only if $v^* = \infty$; and*
- *if F is satisfiable, then there is an f-minimal solution α of F with objective value $f\restriction_\alpha = v^*$.*

We are now ready to give a formal description of the rules in our proof system.

Implicational Derivation Rule

If we can exhibit a derivation of the pseudo-Boolean constraint C from $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$ in our (slightly extended) version of cutting planes as described in Section 2 (i.e., in formal notation, if $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \vdash C$), then we can go from the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to the configuration $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ by the *implicational derivation rule*. By the soundness of the cutting planes proof system, this means that $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \models C$, and so (F, f)-validity is preserved, but, more importantly, the cutting planes derivation provides a simple and efficient way for an algorithm to *verify* that this implication holds. This is a key feature of all rules in our proof system—not only are they sound, but the soundness of every rule application can be efficiently verified by checking a simple, syntactic object.

When doing proof logging, the solver would need to specify by which sequence of cutting planes derivation rules C was obtained. For practical purposes, though, it greatly simplifies matters that in many cases the verifier can figure out the required proof details automatically, meaning that the proof logger can just state the desired constraint without any further information. One important example of this is when C is a reverse unit propagation (RUP) constraint with respect to $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$. Another case is when C is literal-axiom-implied by some other constraint.

Objective Bound Update Rule

The *objective bound update rule* allows improving the estimate of what value can be achieved for the objective function f. We can go from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v')$ if we know an assignment α satisfying $\mathscr{C}$ such that $f\restriction_\alpha = v' < v$. When actually doing proof logging, the solver would specify such an assignment α, which would then be checked by the proof verifier (in our case *VeriPB*).

To argue that this rule preserves (F, f)-validity, we note that the last two items are trivially satisfied (they are weaker after applying the rule than before). The first item is satisfied since item 2 guarantees the existence of an α' satisfying F with an objective value that is at least as good as v'. Note that we have no guarantee that α' will be a solution to F. However, although we will not emphasize this point here, it follows from our formal treatment below that the proof system guarantees that such an f-minimal solution α' to the original formula F can be efficiently reconstructed from the proof (where efficiency is measured in the size of the proof).

Redundance-Based Strengthening Rule

The redundance-based strengthening rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, provided that it can be shown that any assignment α that satisfies $\mathscr{C} \cup \mathscr{D}$ can be transformed into another assignment $\alpha' \preceq_f \alpha$ that satisfies both $\mathscr{C} \cup \mathscr{D}$ and C (in case $\mathscr{O}_\preceq = \mathscr{O}_\top$, the condition $\alpha' \preceq_f \alpha$ just means that $f\restriction_{\alpha'} \leq f\restriction_\alpha$). This rule is borrowed from [Gocht and Nordström, 2021], which in turn relies heavily on [Heule et al., 2017, Buss and Thapen, 2019]. We extend this rule here from decision problems to optimization problems in the natural way.

Formally, we say that C can be derived from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by *redundance-based strengthening*, or just *redundance* for brevity, if there is a substitution ω (which we will refer to as the *witness*) such that

$$\begin{aligned} &\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \\ &\quad (\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \ . \end{aligned} \tag{5}$$

Intuitively, (5) says that if some assignment α satisfies $\mathscr{C} \cup \mathscr{D}$ but falsifies C, then $\alpha' = \alpha \circ \omega$ still satisfies $\mathscr{C} \cup \mathscr{D}$ and also satisfies C. In addition, the condition $f\restriction_\omega \leq f$ ensures that $\alpha \circ \omega$ achieves an objective function value that is at least as good as that for α. This together with the constraints $\mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$ guarantees that $\alpha' \preceq_f \alpha$. For proof logging purposes, the witness ω as well as any non-immediate cutting planes derivations of constraints on the right-hand side of (5) would have to be specified, but, e.g., all RUP constraints or literal-axiom-implied constraints can be left to the verifier to check.

Proposition 3. *If C is derivable from an (F, f)-valid configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by redundance-based strengthening, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ is (F, f)-valid as well.*

Deletion Rule

We also need to be able to delete previously derived constraints. From a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ we can transition to $(\mathscr{C}', \mathscr{D}', \mathscr{O}_\preceq, \vec{z}, v)$ using the *deletion rule* if

1. $\mathscr{D}' \subseteq \mathscr{D}$ and
2. $\mathscr{C}' = \mathscr{C}$ or $\mathscr{C}' = \mathscr{C} \backslash \{C\}$ for some constraint C derivable via the redundance rule from $(\mathscr{C}', \emptyset, \mathscr{O}_\preceq, \vec{z}, v)$.

This last condition above perhaps seems slightly odd, but it is there since deleting arbitrary constraints could violate (F, f)-validity in two different ways. Firstly, it would allow finding better-than-optimal solutions. Secondly, and perhaps surprisingly, in combination with the dominance-based strengthening rule, which we will discuss below, arbitrary deletion is unsound, as it can turn satisfiable instances into unsatisfiable ones. This is illustrated in Example 5 further below.

To see that deletion preserves (F, f)-validity, it is clear that item 1 remains satisfied by deletion, as does the direction of item 2 that claims satisfiability of $\mathscr{C} \cup \{f \leq v'\}$. For the other direction of item 2 and for item 3, intuitively the redundance rule guarantees that solutions of the configuration after deletion can be mapped to solutions of the configuration before deletion that are at least as good.

An alternative to condition 2 would be to enforce the more restrictive demand $\mathscr{C}' \vdash \mathscr{C}$. However, this would prevent the use of some SAT preprocessing techniques such as bounded variable elimination [Eén and Biere, 2005].

Transfer Rule

Constraints can always be moved from the derived set $\mathscr{D}$ to the core set $\mathscr{C}$ using the *transfer rule*, which allows a transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}', \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ if $\mathscr{C} \subseteq \mathscr{C}' \subseteq \mathscr{C} \cup \mathscr{D}$. This clearly preserves (F, f)-validity.

The transfer rule together with deletion allows replacing constraints in the original formula with stronger constraints. For example, assume that $x + y \geq 1$ is in $\mathscr{C}$ and that we derive $x \geq 1$. Then we can move $x \geq 1$ from $\mathscr{D}$ to $\mathscr{C}$ and then delete $x + y \geq 1$. The required redundance check $\{x \geq 1, \neg(x + y \geq 1)\} \vdash \bot$ is immediate.

The rules discussed so far do not change $\mathscr{O}_\preceq$, and so any derivation using these rules only will operate with the trivial preorder $\mathscr{O}_\top$ imposing no conditions. The proof system defined in terms of these rules is a

straightforward extension of *VeriPB* as developed in [Elffers et al., 2020, Gocht et al., 2020b,a, Gocht and Nordström, 2021] to an optimization setting. We next discuss the main contribution of this paper, namely the new dominance rule making use of the preorder $\mathscr{O}_{\preceq}$.

Dominance-Based Strengthening Rule

Any preorder $\preceq$ induces a strict order $\prec$ defined by $\alpha \prec \beta$ if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. The relation $\prec_f$ obtained in this way from the preorder (4) coincides with what Chu and Stuckey [2015] call a *dominance relation* in the context of constraint optimisation. Our dominance rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, similar to the redundance rule. However, for the dominance rule an assignment α satisfying $\mathscr{C} \cup \mathscr{D}$ but falsifying C need only to be mapped to an assignment α' that satisfies $\mathscr{C}$, but not necessarily $\mathscr{D}$ or C. On the other hand, the new assignment α' should satisfy the strict inequality $\alpha' \prec_f \alpha$ and not just $\alpha' \preceq_f \alpha$ as in the redundance rule. To show that this new dominance rule preserves (F, f)-validity, we will prove that it is possible to construct an assignment that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$ by iteratively applying the witness of the dominance rule, in combination with (F, f)-validity of the configuration before application of the dominance rule. As our base case, if α' satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$, we are done. Otherwise, since α' satisfies $\mathscr{C}$, by (F, f)-validity we are guaranteed the existence of an assignment α'' satisfying $\mathscr{C} \cup \mathscr{D}$ for which $\alpha'' \prec_f \alpha' \prec_f \alpha$ holds. If α'' still does not satisfy C, we can repeat the argument. In this way, we get a strictly decreasing sequence (with respect to $\prec_f$) of assignments. Since the set of possible assignments is finite, this sequence will eventually terminate.

Formally, we can derive C by dominance-based strengthening provided that there exists a substitution ω such that

$$\begin{aligned} &\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \\ &\quad \mathscr{C}\restriction_\omega \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \cup \{f\restriction_\omega \ \leq f\} \ , \end{aligned} \tag{6}$$

where $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ and $\neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ together state that $\alpha \circ \omega \prec \alpha$ for any assignment α. A minor technical problem is that the pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ may contain multiple constraints, so that the negation of it is no longer a PB formula. To get around this, we split (6) into two separate conditions and shift $\neg\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega)$ to the premise of the implication, which eliminates the negation. Thus, the formal version of our *dominance-based strengthening rule*, or just *dominance rule* for brevity, says that we can go from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ if there is a substitution ω such that the conditions

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \ \leq f\} \tag{7a}$$

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{7b}$$

are satisfied. Just as for the redundance rule, the witness ω as well as any non-immediate derivations would have to be specified in the proof log.

Proposition 4. *If C is derivable from an (F, f)-valid configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by dominance-based strengthening, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is also (F, f)-valid.*

When introducing the deletion rule, we already mentioned that deleting arbitrary constraints can be unsound in combination with dominance-based strengthening. We now illustrate this phenomenon.

Example 5. *Consider the formula $F = \{p \geq 1\}$ with objective $f \doteq 0$ and the configuration*

$$(\mathscr{C}_1 = \{p \geq 1\}, \mathscr{D}_1 = \{p \geq 1\}, \mathscr{O}_{\preceq}, \{p\}, \infty) \ , \tag{8}$$

where $\mathscr{O}_{\preceq}(u, v)$ is defined as $\{v + \overline{u} \geq 1\}$. This configuration is (F, f)-valid and $\mathscr{C} \cup \mathscr{D}$ is satisfiable. If *we were allowed to delete constraints arbitrarily from $\mathscr{C}$, we could derive a configuration with $\mathscr{C}_2 = \emptyset$ and*

$\mathscr{D}_2 = \{p \geq 1\}$. *However, now the dominance rule can derive* $C \doteq \overline{p} \geq 1$, *using the witness* $\omega = \{p \mapsto 0\}$. *To see that all conditions for applying dominance-based strengthening are indeed satisfied, we notice that conditions* (7a)–(7b) *simplify to*

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \vdash \emptyset \cup \{p + 1 \geq 1\} \cup \emptyset \tag{9a}$$

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \cup \{0 + \overline{p} \geq 1\} \vdash \bot \tag{9b}$$

respectively. Both claims clearly hold, meaning that we arrive at a configuration that contains both $p \geq 1$ *and* $\overline{p} \geq 1$.

Preorder Encodings

As mentioned before, $\mathscr{O}_{\preceq}$ is shorthand for a pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ over two sets of formal placeholder variables $\vec{u} = \{u_1, \ldots, u_n\}$ and $\vec{v} = \{v_1, \ldots, v_n\}$ of equal size, which should also match the size of $\vec{z}$ in the configuration. To use $\mathscr{O}_{\preceq}$ in a proof, it is required to show that this formula encodes a preorder. This is done by providing (in a proof preamble) cutting planes derivations establishing

$$\emptyset \vdash \mathscr{O}_{\preceq}(\vec{u}, \vec{u}) \tag{10a}$$

$$\mathscr{O}_{\preceq}(\vec{u}, \vec{v}) \cup \mathscr{O}_{\preceq}(\vec{v}, \vec{w}) \vdash \mathscr{O}_{\preceq}(\vec{u}, \vec{w}) \tag{10b}$$

where (10a) formalizes reflexivity and (10b) transitivity (and where notation like $\mathscr{O}_{\preceq}(\vec{v}, \vec{w})$ is shorthand for applying to $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ the substitution ω that maps u_i to v_i and v_i to w_i, as discussed in Section 2). These two conditions guarantee that the relation $\preceq$ defined by $\alpha \preceq \beta$ if $\mathscr{O}_{\preceq}(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ forms a preorder on the set of assignments.

By way of example, to encode the lexicographic order $u_1 u_2 \ldots u_n \preceq_{\text{lex}} v_1 v_2 \ldots v_n$, we can use a single constraint

$$\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{v}) \doteq \textstyle\sum_{i=1}^{n} 2^{n-i} \cdot (v_i - u_i) \geq 0 \ . \tag{11}$$

Reflexivity is vacuously true since $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{u}) \doteq 0 \geq 0$, and transitivity also follows easily since adding $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{v})$ and $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{v}, \vec{w})$ yields $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{w})$.

A potential concern with encodings such as (11) is that coefficients can become very large as the number of variables in the order grows. It is perfectly possible to address this by allowing order encodings using auxiliary variables in addition to $\vec{u}$ and $\vec{v}$. We have chosen not to develop the theory for this in the current paper, however, since we feel that it makes the exposition unnecessarily complicated without adding anything of real significance to the scientific contribution.

Order Change Rule

The final proof rule that we need is a rule for introducing a nontrivial order, and it turns out that it can also be convenient to be able to use different orders at different points in the proof. Switching orders is possible, but to maintain soundness it is important to first clear the set $\mathscr{D}$ (after transferring the constraints we want to keep to $\mathscr{C}$). The reason for this is simple: if we allow arbitrary order changes, then the third item of (F, f)-validity would no longer hold, but when $\mathscr{D} = \emptyset$, it is trivially true.

Formally, provided that $\mathscr{O}_{\preceq^2}$ has been established to be a preorder (via cutting planes proofs for (10a) and (10b)), and provided that $\vec{z}_2$ is a list of variables of the size required by this order, it is allowed to go from the configuration $(\mathscr{C}, \emptyset, \mathscr{O}_{\preceq^1}, \vec{z}_1, v)$ to the configuration $(\mathscr{C}, \emptyset, \mathscr{O}_{\preceq^2}, \vec{z}_2, v)$ using the *order change* rule. As explained above, it is clear that this rule preserves (F, f)-validity.

This concludes the presentation of our proof system. Each rule has been shown to preserve (F, f)-validity, and the initial configuration is clearly (F, f)-valid. Therefore, by Theorem 2 our proof system is sound: whenever we can derive a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ such that $\mathscr{C} \cup \mathscr{D}$ contains $0 \geq 1$, it holds that v is the value of f in any f-minimal solution of F (or, for a decision problem, we have $v < \infty$ precisely when F is satisfiable). As mentioned above, in this case the full sequence of configurations $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ together with annotations about the derivation steps—including, in particular, any witnesses ω—contains all information needed to efficiently reconstruct such an f-minimal solution of F. It is also straightforward to show that our proof system is complete: after using the bound update rule to log an optimal solution v^*, it follows from the implicational completeness of cutting planes that contradiction can be derived from $F \cup \{f \leq v^* - 1\}$.

4 Applications

We now exhibit three applications that have not previously admitted efficient certification, and demonstrate that our new method can support simple, practical proof logging in each case. We first show that, by enhancing the *BreakID* tool for SAT solving [Devriendt et al., 2016] with *VeriPB* proof logging, we can cover the entire solving toolchain when symmetries are involved. We then revisit the Crystal Maze example from the introduction. Finally, we discuss how dominance-based strengthening can be used to support vertex domination reasoning in a maximum clique solver. All code for our implementations and experiments, as well as data and scripts for all plots, can be found at https://doi.org/10.5281/zenodo.6373986.

Symmetry Breaking in SAT Solvers

Symmetry handling has a long and successful history in SAT solving, with a wide variety of techniques considered by, e.g., Aloul et al. [2006], Benhamou and Saïs [1994], Benhamou et al. [2010], Devriendt et al. [2012, 2017], Metin et al. [2019], Sabharwal [2009]. These techniques were used to great effect in, e.g., the 2013 and 2016 editions of the *SAT competition*,[2] where the *SAT+UNSAT hard combinatorial track* and the *no-limit* track, respectively, were won by solvers employing symmetry breaking. However, the victory in 2013 can partly be explained by a small parser bug. For reasons such as this, proof logging is now obligatory in the main track of the SAT competition. While it is hard to overemphasize the importance of this development, it unfortunately means that symmetry breaking can no longer be used, since there is no way of efficiently certifying the correctness of such reasoning in *DRAT*. We will now explain how pseudo-Boolean reasoning with the dominance rule can provide proof logging for the *static symmetry breaking* techniques of Devriendt et al. [2016].

Let π be a permutation of the set of literals in a given CNF formula F (i.e., , a bijection on the set of literals), extended to (sets of) clauses in the obvious way. We say that π is a *symmetry* of F if it commutes with negation, i.e., $\pi(\overline{\ell}) = \overline{\pi(\ell)}$, and preserves satisfaction of F, i.e., $\alpha \circ \pi$ satisfies F if and only if α does. A *syntactic symmetry* in addition satisfies that $\pi(F) \doteq F\restriction_{\pi} \doteq F$. As is standard, we only consider syntactic symmetries.

The most common way of breaking symmetries is by adding *lex-leader constraints* [Crawford et al., 1996]. We here use $\preceq_{\text{lex}}$ to denote the lexicographic order on assignments induced by the sequence of variables $x_1, \ldots, x_m$. Given a set G of symmetries of F, a lex-leader constraint is a formula ψ_{LL} such that α satisfies ψ_{LL} if and only if $\alpha \preceq_{\text{lex}} \alpha \circ \pi$ for each $\pi \in G$. Let $\{x_{i_1}, \ldots, x_{i_n}\}$ be the *support* of π (i.e., all

[2] www.satcompetition.org

variables x such that $\pi(x) \neq x$), ordered so that $i_j \leq i_k$ if and only if $j \leq k$. Then the constraints

$$y_0 \geq 1 \tag{12a}$$
$$\overline{y}_{j-1} + \overline{x}_{i_j} + \pi(x_{i_j}) \geq 1 \qquad 1 \leq j \leq n \tag{12b}$$
$$\overline{y}_j + y_{j-1} \geq 1 \qquad 1 \leq j < n \tag{12c}$$
$$\overline{y}_j + \overline{\pi(x_{i_j})} + x_{i_j} \geq 1 \qquad 1 \leq j < n \tag{12d}$$
$$y_j + \overline{y}_{j-1} + \overline{x}_{i_j} \geq 1 \qquad 1 \leq j < n \tag{12e}$$
$$y_j + \overline{y}_{j-1} + \pi(x_{i_j}) \geq 1 \qquad 1 \leq j < n \tag{12f}$$

form a lex-leader constraint for π, where each y_j is a fresh variable representing that α and $\alpha \circ \pi$ are equal up to x_{i_j}, and where (12b) does the actual breaking.

To derive this in our proof system, assume that we have a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{x}, v)$ where assignments are compared lexicographically on $\vec{x} = \{x_1, \ldots, x_m\}$ according to $\mathcal{O}_\preceq$ as in (11). Let π be a syntactic symmetry of $\mathscr{C}$ (i.e., such that $\mathscr{C}\restriction_\pi \doteq \mathscr{C}$) with support contained in $\vec{x}$. In this case

$$C_{LL} \doteq \textstyle\sum_{i=1}^{m} 2^{m-i} \cdot (\pi(x_i) - x_i) \geq 0 \tag{13}$$

expresses that $\pi(\vec{x})$ is greater than or equal to $\vec{x}$. Noting that SAT problems lack an objective function, we can apply the dominance rule with $\omega = \pi$ to derive C_{LL}. To see that (7a) holds, we note that $\neg C_{LL}$ expresses that $\vec{x}$ is strictly larger than $\pi(\vec{x})$, and hence this implies $\mathscr{O}_\preceq(\vec{x}\restriction_\pi, \vec{x})$. Clearly, (7b) is true as well, since its premise contains both C_{LL} and its negation. Since the y-variables are fresh, we can also derive the constraints (12a) and (12c)–(12f) as explained by Gocht and Nordström [2021]. It remains to show how to deduce the constraints (12b) from C_{LL}.

As before, assume that the support of π is $\{x_{i_1}, \ldots, x_{i_n}\}$ with $i_j \leq i_k$ if and only if $j \leq k$. Note first that for all x_i that are not in the support of π, the term $\pi(x_i) - x_i$ disappears since $\pi(x_i) = x_i$ and thus C_{LL} simplifies to

$$\textstyle\sum_{j=1}^{n} 2^{m-i_j} \cdot (\pi(x_{i_j}) - x_{i_j}) \geq 0 \ , \tag{14}$$

which can only hold if the term with the largest coefficient is non-negative. It follows that C_{LL} implies $\pi(x_{i_1}) - x_{i_1} \geq 0$ by reverse unit propagation (RUP), and hence can be derived from our current configuration with the implicational rule, also yielding the weaker constraint (12b) with $j = 1$.

To deal with $j > 1$, we define

$$C_{LL}(0) \doteq C_{LL} \tag{15a}$$
$$C_{LL}(k) \doteq C_{LL}(k-1) + 2^{m-i_k} \cdot (12d\,[j = k]) \tag{15b}$$

where $(12d\,[j = k])$ denotes substitution of j by k in (12d). Simplifying $C_{LL}(k)$ yields

$$\textstyle\sum_{i=1}^{k} 2^{m-i}\overline{y_j} + \sum_{i=k+1}^{m} 2^{m-i} \cdot (\pi(x_i) - x_i) \geq 0 \ , \tag{16}$$

which, in combination with all constraints (12c), directly entails (12b) with $j = k$. To see this, note that if y_k is false, then (12b) is trivially true for $j = k + 1$. On the other hand, if y_k is true, then so are all the preceding y-variables, and the dominant term in $C_{LL}(k)$ becomes $\pi(x_{i_k}) - x_{i_k}$, which implies (12b) for $j = k$ analogously to the case for $j = 1$.

It is important to note here that the order is set once and is the same for all symmetries $\pi \in G$ to be broken. Since constraints are added only to $\mathscr{D}$, dominance rule applications for different symmetries will not

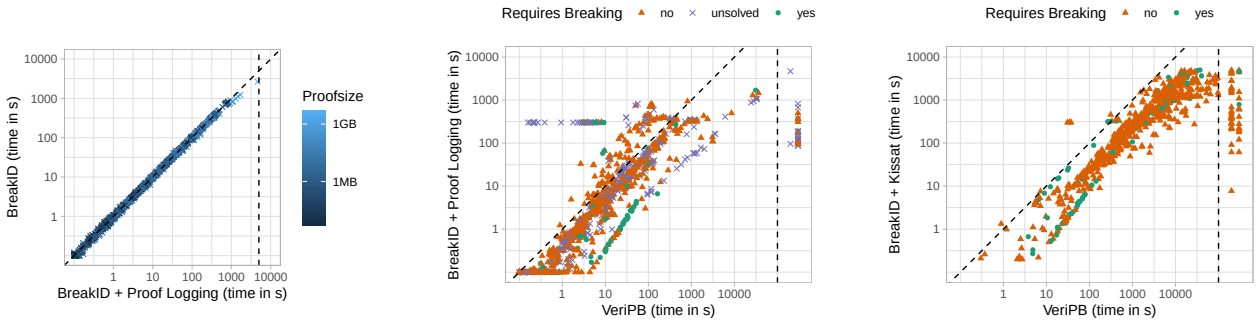


Figure 2: On the left, performance overhead due to proof logging symmetry breaking. In the center, performance of verifying symmetry breaking. On the right, performance of verifying symmetry breaking and SAT solving. Points behind the vertical dashed line indicate timeouts (left) and out of memory (right).

interfere with each other. Furthermore, contrary to the symmetry logging approach of Heule et al. [2015], handling a symmetry once is enough to guarantee complete breaking. See Appendix C for a worked-out *VeriPB* example of symmetry breaking together with explanations of how the proof logging syntax matches rules in our proof system.

To validate our approach, we implemented *VeriPB* proof logging for the symmetry breaking method in *BreakID*, and modified *Kissat*[3] to output *VeriPB*-proofs (since the redundance rule is a generalization of the RAT rule, this required only minor changes). We used a simpler version of the deletion rule that only guarantees to prove a lower bound on the objective value—if this lower bound is infinity, this certifies that decision problems are unsatisfiable (see the discussion of *weak* (F, f)*-validity* in Appendix B).

Out of all the benchmark instances from all the SAT competitions since 2016, we selected all instances in which at least one symmetry was detected; there were 1089 such instances in total. We performed our experiments on machines with dual Intel Xeon E5-2697A v4 processors with 512GBytes RAM and solid-state drive (SSD), running Ubuntu 20.04. We ran twenty instances in parallel on each machine, limiting each instance to 16GBytes RAM, and with a timeout of 5,000s for solving and 100,000s for verification.

The left plot in Figure 2 displays the performance overhead for symmetry breaking, comparing for each instance the running time with and without proof logging. For most instances, the overhead is negligible (99% of instances are at most 32% slower). The other two plots in Figure 2 display the relationship between the time needed to generate a proof (both for SAT and UNSAT instances) and to verify the correctness of this proof. When only considering verification of the symmetry breaking (middle plot), 1058 instances out of 1089 could be verified, 2 timed out, and 29 terminated due to running out of memory. 75% of the instances could be verified within 3.2 times the solving time and 95% within a factor 20. The time needed for verification is thus considerably longer than solving time, but still practical in the majority of cases. After symmetry breaking, 721 instances could be solved with the SAT solver (right plot) and we could verify 671 instances, while for 33 instances verification timed out and for 17 instances the verifier ran out of memory. Notably, 84 instances could only be solved with symmetry breaking, out of which we could verify 81.

Symmetries in Constraint Programming

In the general setting considered in constraint programming, we must deal with variables with larger (non-Boolean) domains and with rich constraints supported by propagation algorithms. One might think that a

[3] http://fmv.jku.at/kissat/

proof system based upon Boolean variables and linear inequalities would not be suitable for this larger class of problem. However, Elffers et al. [2020] showed how to use *VeriPB* for constraint satisfaction problems by first encoding variables and constraints in pseudo-Boolean form, and then constructing cutting planes proofs to justify the behaviour of propagators such as *alldifferent*. Similarly, the work we present here can also be applied to constraint satisfaction and optimisation problems.

Recall the symmetry breaking constraints proposed for the Crystal Maze puzzle in the introduction. Given the difficulties in knowing which combinations of constraints are valid, it would be desirable if these constraints could be *introduced as part of a proof*, rather than taken as axioms. This would give a modeller immediate feedback as to whether the constraints have been chosen correctly. Our proof system is indeed powerful enough to express all three of the examples we presented, and we have implemented a small tool which can write out the appropriate proof fragments; this allows the entire Crystal Maze example to be verified with *VeriPB*. Interestingly, although symmetries can be broken in different ways in high-level CP models (including through lexicographic and value precedence constraints), when we encode the problem in pseudo-Boolean form these differences largely disappear, and after creating a suitable order we can re-use the SAT techniques just discussed. So, although a full proof-logging constraint solver does not yet exist, we can confidently claim that symmetries no longer block this goal.

Lazy Global Domination in Maximum Clique

Gocht et al. [2020a] showed how *VeriPB* can be used to implement proof logging for a wide range of maximum clique algorithms, observing that the cutting planes proof system is rich enough to justify a wide range of bound and inference functions used by various solvers (despite cutting planes not knowing what a graph or clique is). However, there is one clique-solving technique in the literature that is *not* amenable to cutting planes reasoning. In order to solve problem instances that arise from a distance-relaxed clique-finding problem, McCreesh and Prosser [2016] enhanced their maximum clique algorithm with a *lazy global domination* rule that works as follows. Suppose that the solver has constructed a candidate clique C and is considering to extend C by two vertices v and w, where the neighbourhood of v excluding w is a (non-strict) superset of the neighbourhood of w excluding v. Then if the solver first tries v and rejects it, there is no need to branch on w as well.

In principle, it should be possible to introduce additional constraints justifying this kind of reasoning in advance using redundance-based strengthening, without the need for the full dominance breaking framework in Section 3 (with some technicalities involving consistent orderings for tiebreaking). However, due to the prohibitive cost of computing the full vertex dominance relation in advance, McCreesh and Prosser instead implement a form of *lazy* dominance detection, which only triggers following a backtrack.

To provide proof logging for this, we must instead be able to introduce vertex dominance constraints precisely when they are used. It is hard to see how to achieve this with the redundance rule, but it is possible using dominance-based strengthening: we have implemented this in the proof logging maximum clique solver in [Gocht et al., 2020a], as discussed in more detail in Appendix E.

5 Conclusion

In this paper, we show that the pseudo-Boolean proof logging method in *VeriPB* [Gocht and Nordström, 2021] can be extended with a rule for dominance breaking so as to efficiently certify unlimited symmetry breaking in SAT solving, even when combined with XOR and cardinality reasoning. A natural next question is whether our method is strong enough to capture other techniques such as those used for MaxSAT; several such techniques, such as the *dominating unit-clause rule* [Niedermeier and Rossmanith, 2000] and

group subsumed label elimination [Leivo et al., 2020], appear to be special cases of dominance, making this a promising direction. Our work also contributes towards extending proof logging techniques from SAT to other combinatorial solving paradigms such as constraint programming and dedicated graph solving algorithms.

Acknowledgements

Bart Bogaerts was partially supported by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G070521N Ciaran McCreesh was supported by a Royal Academy of Engineering Research Fellowship. 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. Part of this work was carried out while taking part in the semester program *Satisfiability: Theory, Practice, and Beyond* in the spring of 2021 at the Simons Institute for the Theory of Computing at UC Berkeley.

References

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.

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.

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.

Fadi A. Aloul, Karim A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006. ISSN 0018-9340.

Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP '97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 375–389. Springer, OctoberNovember 1997.

Belaïd Benhamou and Lakhdar Saïs. Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning*, 12(1):89–102, February 1994.

Belaïd Benhamou, Tarek Nabhani, Richard Ostrowski, and Mohamed Réda Saïdi. Enhancing clause learning by symmetry in SAT solvers. In *Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI '10)*, volume 1, pages 329–335, October 2010.

Armin Biere. Tracecheck. http://fmv.jku.at/tracecheck/, 2006.

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.

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.

Teobaldo Bulhões, Ruslan Sadykov, and Eduardo Uchoa. A branch-and-price algorithm for the minimum latency problem. *Computers & Operations Research*, 93:66–78, May 2018.

Samuel R. Buss and Jakob Nordström. Proof complexity and SAT solving. In Biere et al. [2021], chapter 7, pages 233–350.

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

Geoffrey Chu and Peter J. Stuckey. Dominance breaking constraints. *Constraints*, 20(2):155–182, April 2015. Preliminary version in *CP '12*.

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

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.

James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pages 148–159, November 1996.

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 *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017a.

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 *Lecture Notes in Computer Science*, pages 118–135. Springer, 2017b.

Erik L. Demeulemeester and Willy S. Herroelen. *Project Scheduling: A Research Handbook*, volume 49 of *International Series in Operations Research & Management Science*. Kluwer Academic Publishers, 2002.

Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *Proceedings of the 24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '12)*, pages 49–56, November 2012.

Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT '16)*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, July 2016.

Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. 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 83–100. Springer, August 2017.

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.

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.

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.

Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4–5):821–839, July 2011.

Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006.

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.

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.

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 2020a.

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 2020b.

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.

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 (FM-CAD '13)*, pages 181–188, October 2013a.

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 2013b.

Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 591–606. Springer, August 2015.

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.

Maaike Hoogeboom, Wout Dullaert, David Lai, and Daniele Vigo. Efficient neighborhood evaluations for the vehicle routing problem with multiple time windows. *Transportation Science*, 54(2):400–416, 2020. doi: 10.1287/trsc.2019.0912. URL https://doi.org/10.1287/trsc.2019.0912.

Antoine Jouglet and Jacques Carlier. Dominance rules in combinatorial optimization problems. *European Journal of Operational Research*, 212(3):433–444, 2011. doi: 10.1016/j.ejor.2010.11.008. URL https://doi.org/10.1016/j.ejor.2010.11.008.

Marcus Leivo, Jeremias Berg, and Matti Järvisalo. Preprocessing in incomplete maxsat solving. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI '20)*, pages 347–354, 2020. doi: 10.3233/FAIA200112. URL https://doi.org/10.3233/FAIA200112.

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

Ciaran McCreesh and Patrick Prosser. Finding maximum k-cliques faster using lazy global domination. In *Proceedings of the 9th Annual Symposium on Combinatorial Search (SOCS '16)*, pages 72–80, July 2016.

Hakan Metin, Souheib Baarir, and Fabrice Kordon. Composing symmetry propagation and effective symmetry breaking for SAT solving. In *Proceedings of the 11th International NASA Formal Methods Symposium (NFM '19)*, volume 11460 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2019. doi: 10.1007/978-3-030-20652-9_21. URL https://doi.org/10.1007/978-3-030-20652-9_21.

Rolf Niedermeier and Peter Rossmanith. New upper bounds for maximum satisfiability. *Journal of Algorithms*, 36(1):63–88, 2000. doi: 10.1006/jagm.2000.1075. URL https://doi.org/10.1006/jagm.2000.1075.

Ashish Sabharwal. SymChaff: Exploiting symmetry in a structure-aware satisfiability solver. *Constraints*, 14(4):478–505, December 2009. Preliminary version in *AAAI '05*.

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.

Toby Walsh. General symmetry breaking constraints. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP '06)*, volume 4204 of *Lecture Notes in Computer Science*, pages 650–664. Springer, September 2006.

Toby Walsh. Symmetry breaking constraints: Recent results. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI '12)*, pages 2192–2198, July 2012.

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 Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

A Introduction to the Technical Appendices

In this collection of technical appendices, we provide some material that had to be omitted in the main text due to space constraints.

In Appendix B, we provide the missing proofs in the formal exposition of our cutting planes proof system with dominance. In Appendix C, we provide details about how our proof logging technique for SAT symmetry breaking works, including a discussion of various optimizations of the basic symmetry breaking algorithm and how we deal with them. In Appendix D we discuss how the methods developed in our work can be used to certify the correctness of added symmetry breaking constraints in constraint programming, and Appendix E contains a fairly detailed discussion of how to certify vertex dominance breaking in a maximum clique solver.

B A Proof System for Dominance Breaking

In this appendix, we give a full, formal presentation of our proof system for verifying dominance breaking, which we have implemented on top of the tool *VeriPB* as developed in the sequence of papers [Elffers et al., 2020, Gocht et al., 2020b,a, Gocht and Nordström, 2021]. In order to give a self-contained presentation, this section essentially copies the material from Section 3, inserting all formal proofs where they belong.

We remark that for applications it is absolutely crucial not only that the proof system be sound, but that all proofs be efficiently machine-verifiable. There are significant challenges involved in making proof logging and verification efficient, but in this section we mostly ignore these aspects of our work and focus on the theoretical underpinnings.

Our foundation is the cutting planes proof system described in Section 2. However, in a proof in our system for (F, f), where f is a linear objective function to be minimized under the pseudo-Boolean formula F (or where $f \doteq 0$ for decision problems), we also allow strengthening F by adding constraints C that are not implied by the formula. Pragmatically, adding C should be in order as long as we keep some optimal solution, i.e., a satisfying assignment to F that minimizes f, which we will refer to as an *f-minimal solution* of F. We will formalize this idea by allowing the use of an additional pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ that, together with a sequence of variables $\vec{z}$, defines a relation $\alpha \preceq \beta$ to hold between assignments α and β if $\mathscr{O}_{\preceq}(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ evaluates to true. We require (a cutting planes proof) that $\mathscr{O}_{\preceq}$ is such that this defines a preorder, i.e., a reflexive and transitive relation. Adding new constraints C will be valid as long as we guarantee to preserve some f-minimal solution that is also minimal with respect to $\preceq$. In other words, $\preceq$ can be combined with f to define a preorder $\preceq_f$ on assignments by

$$\alpha \preceq_f \beta \quad \text{if} \quad \alpha \preceq \beta \text{ and } f\restriction_\alpha \leq f\restriction_\beta \ , \tag{17}$$

and we require that all derivation steps in the proof should preserve some solution that is minimal with respect to $\preceq_f$. The preorder defined by $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ will only become important once we introduce our new *dominance-based strengthening rule* later in this section. For simplicity, up until that point the reader can assume that the pseudo-Boolean formula is $\mathscr{O}_\top \doteq \emptyset$ inducing the trivial preorder relating all assignments, though all proofs presented below work in full generality for the orders that will be introduced later.

A proof for (F, f) in our proof system consists of a sequence of *proof configurations* $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$, where

- $\mathscr{C}$ is a set of pseudo-Boolean *core constraints*;
- $\mathscr{D}$ is another set of pseudo-Boolean *derived constraints*;

- $\mathscr{O}_{\preceq}$ is a PB formula encoding a preorder and $\vec{z}$ a set of literals on which this preorder will be applied; and
- v is the best value found so far for f.

The initial configuration is $(F, \emptyset, \mathscr{O}_{\top}, \emptyset, \infty)$. The distinction between $\mathscr{C}$ and $\mathscr{D}$ is only relevant when a nontrivial preorder is used; we will elaborate on this when discussing dominance. The intended semantics of f and v is that if $v < \infty$, then there exists a solution α satisfying F such that $f\restriction_{\alpha} \leq v$, and in this case the proof can make use of the constraint $f \leq v - 1$ in the search for better solutions. As long as the optimal solution has not been found, it should hold that f-minimal solutions of $\mathscr{C} \cup \mathscr{D}$ have the same objective value as f-minimal solutions of F. The precise relation is formalized in the notion of *valid configurations* as defined next.

Definition 6. *A configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is (F, f)-*valid *if the following conditions hold:*

1. *If $v < \infty$, then there is a total assignment ρ satisfying F such that $f\restriction_{\rho} \leq v$.*
2. *For every $v' < v$, it holds that the sets $F \cup \{f \leq v'\}$ and $\mathscr{C} \cup \{f \leq v'\}$ are equisatisfiable.*
3. *For every total assignment ρ satisfying the constraints $\mathscr{C} \cup \{f \leq v - 1\}$, there exists a total assignment $\rho' \preceq_f \rho$ satisfying $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v - 1\}$.*

We will show that (F, f)-validity is an invariant of our proof system, i.e., that it is preserved by all derivation rules. Note that the two last items together imply that if the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is such that v is not yet the value of an optimal solution, then f-minimal solutions of F and of $\mathscr{C} \cup \mathscr{D}$ have the same objective value, just as desired.

A proof in our proof system ends when the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v^*)$ is such that $\mathscr{C} \cup \mathscr{D}$ contains contradiction $\bot \doteq 0 \geq 1$. In that case, either $v^* = \infty$ and F is unsatisfiable, or v^* is the optimal value (or $v^* = 0$ for a satisfiable decision problem). We state this as a formal theorem.

Theorem 7. *Let F be a pseudo-Boolean formula and f an objective function. If $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v^*)$ is an (F, f)-valid configuration with $\{0 \geq 1\} \subseteq \mathscr{C} \cup \mathscr{D}$, then*

- *F is unsatisfiable if and only if $v^* = \infty$; and*
- *if F is satisfiable, then there is an f-minimal solution α of F with objective value $f\restriction_{\alpha} = v^*$.*

Proof. If F is unsatisfiable, then we must have $v^* = \infty$ due to item 1 of (F, f)-validity.

If F is satisfiable, let α be an f-optimal assignment of F. We will show that $v^* = f\restriction_{\alpha}$. Clearly, $v^* \geq f\restriction_{\alpha}$, otherwise item 1 of (F, f)-validity would yield a strictly better assignment than α, contradicting optimality. If $v^* > f\restriction_{\alpha}$, then α satisfies $F \cup \{f \leq v^* - 1\}$. Hence, item 2 yields an α' that satisfies $\mathscr{C} \cup \{f \leq v^* - 1\}$ and item 3 an α'' that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v^* - 1\}$, which contradicts the assumption that $0 \geq 1$ is in $\mathscr{C} \cup \mathscr{D}$. It follows that $v^* \leq f\restriction_{\alpha}$ and thus $v^* = f\restriction_{\alpha}$, as desired. □

We are now ready to give a formal description of the rules in our proof system and argue that these rules preserve (F, f)-validity.

Implicational Derivation Rule

If we can exhibit a derivation of the pseudo-Boolean constraint C from $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$ in our (slightly extended) version of cutting planes as described in Section 2 (i.e., in formal notation, if $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \vdash C$), then we can go from the configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ to the configuration $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by the *implicational derivation rule*. By the soundness of the cutting planes proof system, this means that $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\} \models C$, and so (F, f)-validity is preserved, but, more importantly, the cutting planes derivation provides a simple and efficient way for an algorithm to *verify* that this implication holds. This is a key feature of all rules in our proof system—not only are they sound, but the soundness of every rule application can be efficiently verified by checking a simple, syntactic object.

When doing proof logging, the solver would need to specify by which sequence of cutting planes derivation rules C was obtained. For practical purposes, though, it greatly simplifies matters that in many cases the verifier can figure out the required proof details automatically, meaning that the proof logger can just state the desired constraint without any further information. One important example of this is when C is a reverse unit propagation (RUP) constraint with respect to $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v-1\}$. Another case is when C is literal-axiom-implied by some other constraint.

Objective Bound Update Rule

The *objective bound update rule* allows improving the estimate of what value can be achieved for the objective function f. We can go from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v')$ if we know an assignment α satisfying $\mathscr{C}$ such that $f\restriction_\alpha = v' < v$. When actually doing proof logging, the solver would specify such an assignment α, which would then be checked by the proof verifier (in our case *VeriPB*).

To argue that this rule preserves (F, f)-validity, we note that the last two items are trivially satisfied (they are weaker after applying the rule than before). The first item is satisfied since item 2 guarantees the existence of an α' satisfying F with an objective value that is at least as good as v'. Note that we have no guarantee that α' will be a solution to F. However, although we will not emphasize this point here, it follows from our formal treatment below that the proof system guarantees that such an f-minimal solution α' to the original formula F can be efficiently reconstructed from the proof (where efficiency is measured in the size of the proof).

Redundance-Based Strengthening Rule

The redundance-based strengthening rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, provided that it can be shown that any assignment α that satisfies $\mathscr{C} \cup \mathscr{D}$ can be transformed into another assignment $\alpha' \preceq_f \alpha$ that satisfies both $\mathscr{C} \cup \mathscr{D}$ and C (in case $\mathscr{O}_{\preceq} = \mathscr{O}_{\top}$, the condition $\alpha' \preceq_f \alpha$ just means that $f\restriction_{\alpha'} \leq f\restriction_\alpha$). This rule is borrowed from [Gocht and Nordström, 2021], which in turn relies heavily on [Heule et al., 2017, Buss and Thapen, 2019]. We extend this rule here from decision problems to optimization problems in the natural way.

Formally, we say that C can be derived from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by *redundance-based strengthening*, or just *redundance* for brevity, if there is a substitution ω (which we will refer to as the *witness*) such that

$$\begin{aligned} &\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \\ &\quad (\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \leq f\} \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \ . \end{aligned} \tag{18}$$

Intuitively, (18) says that if some assignment α satisfies $\mathscr{C} \cup \mathscr{D}$ but falsifies C, then $\alpha' = \alpha \circ \omega$ still satisfies $\mathscr{C} \cup \mathscr{D}$ and also satisfies C. In addition, the condition $f\restriction_\omega \leq f$ ensures that $\alpha \circ \omega$ achieves an

objective function value that is at least as good as that for α. This together with the constraints $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ guarantees that $\alpha' \preceq_f \alpha$. For proof logging purposes, the witness ω as well as any non-immediate cutting planes derivations of constraints on the right-hand side of (18) would have to be specified, but, e.g., all RUP constraints or literal-axiom-implied constraints can be left to the verifier to check.

Proposition 8. *If C is derivable from an (F, f)-valid configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ by redundance-based strengthening, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z}, v)$ is (F, f)-valid as well.*

Proof. Items 1 and 2 of (F, f)-validity remain satisfied since F, v, and $\mathscr{C}$ are unchanged. Our proof of item 3 extends proofs of similar properties for decision problems [Heule et al., 2017, Buss and Thapen, 2019, Gocht and Nordström, 2021]. Consider an assignment ρ satisfying $\mathscr{C} \cup \{f \leq v - 1\}$. We will construct an assignment $\rho' \preceq_f \rho$, i.e., such that $f\restriction_{\rho'} \leq f\restriction_\rho$ and $\mathscr{O}_{\preceq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_\rho)$ hold, that also satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$.

Without loss of generality (due to item 3 of (F, f)-validity, which holds for the first configuration), we can assume that ρ also satisfies $\mathscr{D}$. If ρ satisfies C, then we use $\rho' = \rho$ and all conditions are satisfied (recall that $\mathscr{O}_{\preceq}$ induces a preorder, and hence a reflexive relation: for any ρ, $\mathscr{O}_{\preceq}(\vec{z}\restriction_\rho, \vec{z}\restriction_\rho)$ holds). Otherwise, choose $\rho' = \rho \circ \omega$. We know that ρ satisfies $\mathscr{C} \cup \mathscr{D} \cup \neg C$, and hence by (18) ρ also satisfies

$$(\mathscr{C} \cup \mathscr{D} \cup C)\restriction_\omega \cup \{f\restriction_\omega \ \leq f\} \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \ . \tag{19}$$

Clearly, for any constraint D, it holds that $(D\restriction_\omega)\restriction_\rho = D\restriction_{\rho\circ\omega}$ and thus if ρ satisfies $D\restriction_\omega$, then $\rho' = \rho \circ \omega$ satisfies D. Therefore, ρ' satisfies $\mathscr{C} \cup \mathscr{D} \cup C$. Additionally, ρ satisfies $\{f\restriction_\omega \ \leq f\}$, and hence $f\restriction_{\rho'} \leq f\restriction_\rho$. Similarly, ρ satisfies $\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z})$ and $(\mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}))\restriction_\rho \doteq \mathscr{O}_{\preceq}(\vec{z}\restriction_{\rho'}, \vec{z}\restriction_\rho)$ thus holds, which concludes our proof. □

Deletion Rule

We also need to be able to delete previously derived constraints. From a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ we can transition to $(\mathscr{C}', \mathscr{D}', \mathscr{O}_{\preceq}, \vec{z}, v)$ using the *deletion rule* if

1. $\mathscr{D}' \subseteq \mathscr{D}$ and
2. $\mathscr{C}' = \mathscr{C}$ or $\mathscr{C}' = \mathscr{C} \backslash \{C\}$ for some constraint C derivable via the redundance rule from $(\mathscr{C}', \emptyset, \mathscr{O}_{\preceq}, \vec{z}, v)$.

This last condition above perhaps seems slightly odd, but it is there since deleting arbitrary constraints could violate (F, f)-validity in two different ways. Firstly, it would allow finding better-than-optimal solutions. Secondly, and perhaps surprisingly, in combination with the dominance-based strengthening rule, which we will discuss below, arbitrary deletion is unsound, as it can turn satisfiable instances into unsatisfiable ones. This is illustrated in Example 12 further below.

To see that deletion preserves (F, f)-validity, it is clear that item 1 remains satisfied by deletion, as does the direction of item 2 that claims satisfiability of $\mathscr{C} \cup \{f \leq v'\}$. Now let α be an assignment that satisfies $\mathscr{C}' \cup \{f \leq v'\}$ for $v' < v$; we use this to construct a satisfying assignment α' for $F \cup \{f \leq v'\}$. If $\mathscr{C}' = \mathscr{C}$, we get α' from the (F, f)-validity of the original configuration, so assume $\mathscr{C}' = \mathscr{C} \setminus \{C\}$. If α satisfies C, it satisfies $\mathscr{C}$, and again the claim follows from (F, f)-validity of the original configuration. Assume therefore that α does not satisfy C. Since C is derivable via redundance from $(\mathscr{C}', \emptyset, \mathscr{O}_{\preceq}, \vec{z}, v)$, it holds that

$$\mathscr{C}' \cup \{\neg C\} \vdash (\mathscr{C}' \cup C)\restriction_\omega \cup \{f\restriction_\omega \ \leq f\} \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_\omega, \vec{z}) \ . \tag{20}$$

This yields an assignment $\alpha'' = \alpha \circ \omega$ satisfying $\mathscr{C} = \mathscr{C}' \cup \{C\}$ such that $f\restriction_{\alpha''} \leq f\restriction_\alpha \leq v'$, showing that $\mathscr{C} \cup \{f \leq v'\}$ is satisfiable. Appealing to the (F, f)-validity of the original configuration, we then find

an α' with $f\restriction_{\alpha'} \leq v'$ that satisfies F, proving that indeed the second item holds. The proof for item 3 is similar: $\alpha \circ \omega$ satisfies $\mathscr{C}' \cup \{C\}$, and applying (F, f)-validity of the original configuration yields an α' with $\alpha' \preceq_f \alpha \circ \omega \preceq_f \alpha$ that satisfies $\mathscr{D}$.

An alternative to condition 2 would be to enforce the more restrictive demand $\mathscr{C}' \vdash \mathscr{C}$. However, this would prevent the use of some SAT preprocessing techniques such as bounded variable elimination [Eén and Biere, 2005].

In practice, checking deletions can make it more difficult to implement proof logging, or could have negative effects on performance. An alternative is to use a more liberal deletion rule, which also allows deleting constraints from $\mathscr{C}$ if $\mathscr{D}$ is empty. In this case, unsatisfiable instances can become satisfiable and better than optimal solutions can be introduced, but we can still verify a lower bound on the best objective value. This means that if the solver provides a solution to the original formula F that matches the verified lower bound, then this solution is guaranteed to be optimal. To prove that the proof system remains sound with this more liberal deletion rule, we need to adjust our invariant.

Definition 9. *A configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is* weakly (F, f)-valid *if the following conditions hold:*

1. *For every $v' < v$, it holds that if $F \cup \{f \leq v'\}$ is satisfiable then $\mathscr{C} \cup \{f \leq v'\}$ is satisfiable.*
2. *For every total assignment ρ satisfying the constraints $\mathscr{C} \cup \{f \leq v - 1\}$, there is a total assignment $\rho' \preceq_f \rho$ satisfying $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v - 1\}$.*

We will only show that each rule preserves (F, f)-validity, because the same proofs can be used to show that weak (F, f)-validity is preserved as well and while deleting from $\mathscr{C}$ if $\mathscr{D}$ is empty does not preserve (F, f)-validity, it is easy to see that weak (F, f)-validity is preserved. With this weaker invariant, we also get a weaker result for the final configuration.

Theorem 10. *Given a formula F and an objective function f, let $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v^*)$ be a weakly (F, f)-valid configuration with $\{0 \geq 1\} \subseteq \mathscr{C} \cup \mathscr{D}$. It holds that*

- *for any solution α of F we have $f\restriction_\alpha \geq v^*$, and especially,*
- *if $v^* = \infty$ then F is unsatisfiable.*

Proof. If F is satisfiable, then let α be a satisfying assignment of F. If $v^* > f\restriction_\alpha$, then α satisfies $F \cup \{f \leq v^* - 1\}$. Hence, item 1 yields an α' that satisfies $\mathscr{C} \cup \{f \leq v^* - 1\}$ and item 2 an α'' that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{f \leq v^* - 1\}$, which contradicts the assumption that $0 \geq 1$ is in $\mathscr{C} \cup \mathscr{D}$. It follows that $v^* \leq f\restriction_\alpha$. □

Transfer Rule

Constraints can always be moved from the derived set $\mathscr{D}$ to the core set $\mathscr{C}$ using the *transfer rule*, which allows a transition from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}', \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ if $\mathscr{C} \subseteq \mathscr{C}' \subseteq \mathscr{C} \cup \mathscr{D}$. This clearly preserves (F, f)-validity.

The transfer rule together with deletion allows replacing constraints in the original formula with stronger constraints. For example, assume that $x + y \geq 1$ is in $\mathscr{C}$ and that we derive $x \geq 1$. Then we can move $x \geq 1$ from $\mathscr{D}$ to $\mathscr{C}$ and then delete $x + y \geq 1$. The required redundance check $\{x \geq 1, \neg(x + y \geq 1)\} \vdash \bot$ is immediate.

The rules discussed so far do not change $\mathscr{O}_\preceq$, and so any derivation using these rules only will operate with the trivial preorder $\mathscr{O}_\top$ imposing no conditions. The proof system defined in terms of these rules is a straightforward extension of *VeriPB* as developed in [Elffers et al., 2020, Gocht et al., 2020b,a, Gocht and Nordström, 2021] to an optimization setting. We next discuss the main contribution of this paper, namely the new dominance rule making use of the preorder $\mathscr{O}_\preceq$.

Dominance-Based Strengthening Rule

Any preorder $\preceq$ induces a strict order $\prec$ defined by $\alpha \prec \beta$ if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. The relation $\prec_f$ obtained in this way from the preorder (17) coincides with what Chu and Stuckey [2015] call a *dominance relation* in the context of constraint optimisation. Our dominance rule allows deriving a constraint C from $\mathscr{C} \cup \mathscr{D}$ even if C is not implied, similar to the redundance rule. However, for the dominance rule an assignment α satisfying $\mathscr{C} \cup \mathscr{D}$ but falsifying C need only to be mapped to an assignment α' that satisfies $\mathscr{C}$, but not necessarily $\mathscr{D}$ or C. On the other hand, the new assignment α' should satisfy the strict inequality $\alpha' \prec_f \alpha$ and not just $\alpha' \preceq_f \alpha$ as in the redundance rule. To show that this new dominance rule preserves (F, f)-validity, we will prove that it is possible to construct an assignment that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$ by iteratively applying the witness of the dominance rule, in combination with (F, f)-validity of the configuration before application of the dominance rule. As our base case, if α' satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$, we are done. Otherwise, since α' satisfies $\mathscr{C}$, by (F, f)-validity we are guaranteed the existence of an assignment α'' satisfying $\mathscr{C} \cup \mathscr{D}$ for which $\alpha'' \prec_f \alpha' \prec_f \alpha$ holds. If α'' still does not satisfy C, we can repeat the argument. In this way, we get a strictly decreasing sequence (with respect to $\prec_f$) of assignments. Since the set of possible assignments is finite, this sequence will eventually terminate.

Formally, we can derive C by dominance-based strengthening provided that there exists a substitution ω such that

$$\begin{aligned} &\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \\ &\quad \mathscr{C}\restriction_\omega \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \cup \neg\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega) \cup \{f\restriction_\omega \leq f\} \ , \end{aligned} \tag{21}$$

where $\mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$ and $\neg\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$ together state that $\alpha \circ \omega \prec \alpha$ for any assignment α. A minor technical problem is that the pseudo-Boolean formula $\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$ may contain multiple constraints, so that the negation of it is no longer a PB formula. To get around this, we split (21) into two separate conditions and shift $\neg\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$ to the premise of the implication, which eliminates the negation. Thus, the formal version of our *dominance-based strengthening rule*, or just *dominance rule* for brevity, says that we can go from $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ to $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ if there is a substitution ω such that the conditions

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\} \tag{22a}$$

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{22b}$$

are satisfied. Just as for the redundance rule, the witness ω as well as any non-immediate derivations would have to be specified in the proof log.

Proposition 11. *If C is derivable from an (F, f)-valid configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ by dominance-based strengthening, then $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_\preceq, \vec{z}, v)$ is also (F, f)-valid.*

Proof. The first two items of (F, f)-validity are clearly satisfied, since F, $\mathscr{C}$, and v are unchanged. Assume towards contradiction that the last item *does not* hold. Let S denote the set of assignments α that (1) satisfy $\mathscr{C} \cup \{f \leq v - 1\}$ and (2) admit no $\alpha' \preceq_f \alpha$ satisfying $\mathscr{C} \cup \mathscr{D} \cup \{C\}$. By our assumption, S is non-empty.

Let α be some $\prec_f$-minimal assignment in S. Since $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z}, v)$ is (F, f)-valid, there exists some $\alpha_1 \preceq \alpha$ that satisfies $\mathscr{C} \cup \mathscr{D}$. We know that α_1 cannot satisfy C since $\alpha \in S$. Hence, α_1 satisfies $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\}$. From (22a) it follows that α_1 satisfies $\mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \leq f\}$ and thus that $\mathscr{O}_\preceq(\vec{z}\restriction_{\alpha_1 \circ \omega}, \vec{z}\restriction_{\alpha_1})$ and $f\restriction_{\alpha_1 \circ \omega} \leq f\restriction_{\alpha_1}$ are satisfied. In other words, $\alpha_1 \circ \omega \preceq_f \alpha_1$. By (22b), it follows that α_1 does not satisfy $\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$, i.e., $\mathscr{O}_\preceq(\vec{z}\restriction_{\alpha_1}, \vec{z}\restriction_{\alpha_1 \circ \omega})$ does not hold and thus $\alpha_1 \not\preceq_f \alpha_1 \circ \omega$. Now let α_2 be $\alpha_1 \circ \omega$. We showed that $\alpha_2 \prec_f \alpha_1 \preceq_f \alpha$. Furthermore, since α_1 satisfies $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\}$, (7a) yields that α_2 satisfies $\mathscr{C}$. Thus α_2 satisfies $\mathscr{C} \cup \{f \leq v - 1\}$. Since $\alpha_2 \prec_f \alpha$, and α is a minimal element of S, it cannot be that $\alpha_2 \in S$. Thus, there must exist a $\alpha' \preceq_f \alpha_2$ that satisfies $\mathscr{C} \cup \mathscr{D} \cup \{C\}$. However, it is also so that $\alpha' \preceq_f \alpha$,

and since $\alpha \in S$ this means that α' cannot satisfy $\mathscr{C} \cup \mathscr{D} \cup \{C\}$. This yields a contradiction, thereby finishing our proof. □

When introducing the deletion rule, we already mentioned that deleting arbitrary constraints can be unsound in combination with dominance-based strengthening. We now illustrate this phenomenon.

Example 12. *Consider the formula* $F = \{p \geq 1\}$ *with objective* $f \doteq 0$ *and the configuration*

$$(\mathscr{C}_1 = \{p \geq 1\}, \mathscr{D}_1 = \{p \geq 1\}, \mathscr{O}_{\preceq}, \{p\}, \infty) \ , \tag{23}$$

where $\mathscr{O}_{\preceq}(u, v)$ *is defined as* $\{v + \overline{u} \geq 1\}$. *This configuration is* (F, f)*-valid and* $\mathscr{C} \cup \mathscr{D}$ *is satisfiable.* If *we were allowed to delete constraints arbitrarily from* $\mathscr{C}$*, we could derive a configuration with* $\mathscr{C}_2 = \emptyset$ *and* $\mathscr{D}_2 = \{p \geq 1\}$. *However, now the dominance rule can derive* $C \doteq \overline{p} \geq 1$*, using the witness* $\omega = \{p \mapsto 0\}$. *To see that all conditions for applying dominance-based strengthening are indeed satisfied, we notice that conditions* (7a)–(7b) *simplify to*

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \vdash \emptyset \cup \{p + 1 \geq 1\} \cup \emptyset \tag{24a}$$

$$\emptyset \cup \{p \geq 1\} \cup \{p \geq 1\} \cup \{0 + \overline{p} \geq 1\} \vdash \bot \tag{24b}$$

respectively. Both claims clearly hold, meaning that we arrive at a configuration that contains both $p \geq 1$ *and* $\overline{p} \geq 1$.

Preorder Encodings

As mentioned before, $\mathscr{O}_{\preceq}$ is shorthand for a pseudo-Boolean formula $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ over two sets of formal placeholder variables $\vec{u} = \{u_1, \ldots, u_n\}$ and $\vec{v} = \{v_1, \ldots, v_n\}$ of equal size, which should also match the size of $\vec{z}$ in the configuration. To use $\mathscr{O}_{\preceq}$ in a proof, it is required to show that this formula encodes a preorder. This is done by providing (in a proof preamble) cutting planes derivations establishing

$$\emptyset \vdash \mathscr{O}_{\preceq}(\vec{u}, \vec{u}) \tag{25a}$$

$$\mathscr{O}_{\preceq}(\vec{u}, \vec{v}) \cup \mathscr{O}_{\preceq}(\vec{v}, \vec{w}) \vdash \mathscr{O}_{\preceq}(\vec{u}, \vec{w}) \tag{25b}$$

where (25a) formalizes reflexivity and (25b) transitivity (and where notation like $\mathscr{O}_{\preceq}(\vec{v}, \vec{w})$ is shorthand for applying to $\mathscr{O}_{\preceq}(\vec{u}, \vec{v})$ the substitution ω that maps u_i to v_i and v_i to w_i, as discussed in Section 2). These two conditions guarantee that the relation $\preceq$ defined by $\alpha \preceq \beta$ if $\mathscr{O}_{\preceq}(\vec{z}\restriction_\alpha, \vec{z}\restriction_\beta)$ forms a preorder on the set of assignments.

By way of example, to encode the lexicographic order $u_1 u_2 \ldots u_n \preceq_{\text{lex}} v_1 v_2 \ldots v_n$, we can use a single constraint

$$\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{v}) \doteq \textstyle\sum_{i=1}^{n} 2^{n-i} \cdot (v_i - u_i) \geq 0 \ . \tag{26}$$

Reflexivity is vacuously true since $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{u}) \doteq 0 \geq 0$, and transitivity also follows easily since adding $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{v})$ and $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{v}, \vec{w})$ yields $\mathscr{O}_{\preceq_{\text{lex}}}(\vec{u}, \vec{w})$ (where we tacitly assume that the constraint resulting from this addition is implicitly simplified by collecting like terms, performing any cancellations, and shifting any constants to the right-hand side of the inequality, as mentioned in Section 2).

A potential concern with encodings such as (26) is that coefficients can become very large as the number of variables in the order grows. It is perfectly possible to address this by allowing order encodings using auxiliary variables in addition to $\vec{u}$ and $\vec{v}$. We have chosen not to develop the theory for this in the current paper, however, since we feel that it makes the exposition unnecessarily complicated without adding anything of real significance to the scientific contribution.

Order Change Rule

The final proof rule that we need is a rule for introducing a nontrivial order, and it turns out that it can also be convenient to be able to use different orders at different points in the proof. Switching orders is possible, but to maintain soundness it is important to first clear the set $\mathscr{D}$ (after transferring the constraints we want to keep to $\mathscr{C}$). The reason for this is simple: if we allow arbitrary order changes, then the third item of (F, f)-validity would no longer hold, but when $\mathscr{D} = \emptyset$, it is trivially true.

Formally, provided that $\mathscr{O}_{\preceq^2}$ has been established to be a preorder (via cutting planes proofs for (25a) and (25b)), and provided that $\vec{z}_2$ is a list of variables of the size required by this order, it is allowed to go from the configuration $(\mathscr{C}, \emptyset, \mathscr{O}_{\preceq^1}, \vec{z}_1, v)$ to the configuration $(\mathscr{C}, \emptyset, \mathscr{O}_{\preceq^2}, \vec{z}_2, v)$ using the *order change* rule. As explained above, it is clear that this rule preserves (F, f)-validity.

This concludes the presentation of our proof system. Each rule has been shown to preserve (F, f)-validity, and the initial configuration is clearly (F, f)-valid. Therefore, by Theorem 7 our proof system is sound: whenever we can derive a configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ such that $\mathscr{C} \cup \mathscr{D}$ contains $0 \geq 1$, it holds that v is the value of f in any f-minimal solution of F (or, for a decision problem, we have $v < \infty$ precisely when F is satisfiable). As mentioned above, in this case the full sequence of configurations $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z}, v)$ together with annotations about the derivation steps—including, in particular, any witnesses ω—contains all information needed to efficiently reconstruct such an f-minimal solution of F. It is also straightforward to show that our proof system is complete: after using the bound update rule to log an optimal solution v^*, it follows from the implicational completeness of cutting planes that contradiction can be derived from $F \cup \{f \leq v^* - 1\}$.

C Technical appendix on Symmetry Breaking in SAT Solvers

In the "Symmetry Breaking in SAT Solvers" subsection, we discuss the core ideas that underlie most modern symmetry breaking tools for SAT. Devriendt et al. [2016] extend these ideas further in a couple of ways. In this technical appendix, we briefly discuss these techniques and how and why they fit in our proof system.

The most important contribution of Devriendt et al. [2016] is detecting so-called *row interchangeability*. The goal of this optimization is to not just take an arbitrary set of generators of the symmetry group and an arbitrary lexicographic order, but to choose "the right" set of generators and "the right" variable order (with which to define the lexicographic order). Devriendt et al. [2016] showed that for groups that exhibit a certain structure, breaking symmetries of a good set of *generators*, with a matching order, can guarantee that the entire symmetry *group* is broken completely. Since our logging techniques simply use the same lexicographic order as the breaking tool, and work for an arbitrary generator set, this automatically works with the techniques described in Section 4.

Another (optional) modification *BreakID* implements is writing out a more *compact encoding*. The authors observed that the definitions the y-variables can be weakened: the clauses (12c) and (12d) in the "Symmetry Breaking in SAT Solvers" subsection can be omitted. Since our definition of $C_{LL}(k)$ uses these clauses, we cannot simply omit them in our proof. However, all the symmetry breaking constraints are added in the set $\mathscr{D}$, and so we can remove these constraints from $\mathscr{D}$ as soon as they are no longer needed for the proof logging derivations.

Next, *BreakID* has an optimization based on *stabilizer subgroups* to detect a plethora of binary clauses. Since these binary clauses are all clauses of the form (12b) with $j = 1$, the described proof logging techniques also work for this optimization, provided we keep track of which symmetry is used for each such binary clause. However, *BreakID* currently does no such bookkeeping. While it is in principle possible to do so, we did not implement this yet.

Finally, *BreakID* supports *partial symmetry breaking*. That is, instead of adding the constraints (12b)–(12f) for *every* j, this is only done for $j < L$ with L a limit that can be chosen by the user. The reasoning behind this is that the larger j, gets, the weaker the added breaking constraint is. By only doing this, for instance for $j < 100$, the size of the added constraints can get significantly smaller without losing too much breaking power. Since we only need to do proof logging for the clauses that are actually added by *BreakID*, this optimization works out-of-the-box. However, there is an important caveat here: in benchmarks where there are huge symmetries, e.g., symmetries permuting all the variables in the problem, even when this optimisation is used, a naive implementation of our proof logging technique suffers from serious performance problems. The reason is that in principle the order $\mathscr{O}_{\preceq}$ is defined on all variables that are permuted by the symmetries. If there are many such variables, this order in itself can get huge (the largest coefficient is exponentially large in terms of the number of variables). Luckily, there is a simple solution to this problem, namely not taking $\vec{x}$ to be the set of all variables that are permuted, but only the set of variables on which we will actually do breaking (for each symmetry, the first L variables in its support); this solution was implemented in the experiments we presented.

A Complete Example of Proof Logging Symmetry Breaking

We now present a complete example of proof logging for symmetry for the well-known pigeon-hole problem. We consider an instance of this problem with 4 pigeons and 3 holes. We use variables p_{ij} to represent that pigeon i resides in hole j. The input for the symmetry breaking preprocessor consists of the constraints

$$p_{11} + p_{12} + p_{13} \geq 1 \tag{C1}$$

$$p_{21} + p_{22} + p_{23} \geq 1 \tag{C2}$$

$$p_{31} + p_{32} + p_{33} \geq 1 \tag{C3}$$

$$p_{41} + p_{42} + p_{43} \geq 1 \tag{C4}$$

$$\overline{p_{11}} + \overline{p_{21}} \geq 1 \tag{C5}$$

$$\overline{p_{11}} + \overline{p_{31}} \geq 1 \tag{C6}$$

$$\overline{p_{11}} + \overline{p_{41}} \geq 1 \tag{C7}$$

$$\overline{p_{21}} + \overline{p_{31}} \geq 1 \tag{C8}$$

$$\overline{p_{21}} + \overline{p_{41}} \geq 1 \tag{C9}$$

$$\overline{p_{31}} + \overline{p_{41}} \geq 1 \tag{C10}$$

$$\overline{p_{12}} + \overline{p_{22}} \geq 1 \tag{C11}$$

$$\overline{p_{12}} + \overline{p_{32}} \geq 1 \tag{C12}$$

$$\overline{p_{12}} + \overline{p_{42}} \geq 1 \tag{C13}$$

$$\overline{p_{22}} + \overline{p_{32}} \geq 1 \tag{C14}$$

$$\overline{p_{22}} + \overline{p_{42}} \geq 1 \tag{C15}$$

$$\overline{p_{32}} + \overline{p_{42}} \geq 1 \tag{C16}$$

$$\overline{p_{13}} + \overline{p_{23}} \geq 1 \tag{C17}$$

$$\overline{p_{13}} + \overline{p_{33}} \geq 1 \tag{C18}$$

$$\overline{p_{13}} + \overline{p_{43}} \geq 1 \tag{C19}$$

$$\overline{p_{23}} + \overline{p_{33}} \geq 1 \tag{C20}$$

$$\overline{p_{23}} + \overline{p_{43}} \geq 1 \tag{C21}$$

$$\overline{p_{33}} + \overline{p_{43}} \geq 1 \tag{C22}$$

where the first four constraints represent that each pigeon resides in at least one hole, and the rest that each hole is occupied by at most one pigeon.

Introducing the order

A *VeriPB* proof starts with a proof header (stating which version of the proof system is used) and an instruction to load the input formula

```
pseudo-Boolean proof version 1.2
f 22
```

where the 22 is the number of formulas in the input (to ensure consistent constraint numbering). To do symmetry breaking, the proof *BreakID* yields, then contains the definition of the pre-order

```
pre_order exp22
  vars
    left u1 u2 u3 u4 u5 u6 u7 u8 u9 u10 u11 u12
    right v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12
    aux
  end

  def
    -1 u12 1 v12 -2 u11 2 v11 -4 u10 4 v10 -8 u9 8 v9 -16 u8 16 v8 -32 u7 32 v7 -64
        ↪ u6 64 v6 -128 u5 128 v5 -256 u4 256 v4 -512 u3 512 v3 -1024 u2 1024 v2
        ↪ -2048 u1 2048 v1 >= 0;
```

```
L12     end
L13
L14     transitivity
L15       vars
L16         fresh_right w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12
L17       end
L18       proof
L19         proofgoal #1
L20           p 1 2 + 3 +
L21           c -1
L22         qed
L23       qed
L24     end
L25   end
```

The pre-order is given a name (`exp22`) in Line 3. Lines 5 and 6 introduce two times twelve auxiliary variables to define the order over. Line 11 then provides the well-known exponential encoding of the fact that the `u`-variables are lexicographically smaller than the `v`-variables. To prove transitivity, another set of variables (called `w`) is introduced. Formally, we need to show that equation (10b) holds. To prove this, we assume $\mathscr{O}_{\preceq}(\vec{u},\vec{v})$, $\mathscr{O}_{\preceq}(\vec{v},\vec{w})$ and $\neg\mathscr{O}_{\preceq}(\vec{u},\vec{w})$ hold. When instantiated with the specified order, these three constraints are

$$-u_{12}+v_{12}-2u_{11}+2v_{11}-4u_{10}+4v_{10}-\cdots\geq 0 \tag{T1}$$

$$-v_{12}+w_{12}-2v_{11}+2w_{11}-4v_{10}+4w_{10}-\cdots\geq 0 \tag{T2}$$

$$u_{12}-w_{12}+2u_{11}-2w_{11}+4u_{10}-4w_{10}+\cdots\geq -1, \tag{T3}$$

where we use the T-numbering to emphasize that these are not constraints learned in the proof system, but temporary constraints, local to the proof of transitivity. Line 20 is an instruction to add constraints (T1–T3), resulting (after simplification) in the constraint

$$0\geq -1, \tag{T4}$$

Line 21 then states that the last derived constraint (the -1 stands for the last derived constraint) is a conflicting constraint, thereby concluding the proof of transitivity. Note that no proof for reflexivity is given since for simple orders such as the one specified here, veriPB's autoproving can construct a proof itself.

The proof continues with the instruction

```
L26   load_order exp22 p21 p22 p23 p11 p12 p13 p31 p32 p33 p41 p42 p43
```

stating that the order should be instantiated with the variables from the input. Do note that in the the chosen instantiation, (all variables related to) pigeon 2 are ordered before pigeon 1, then pigeons 3 and 4. In other words, in the lex-leader order, pigeon 2 has the highest importance.

Logging the breaking of a first symmetry

The next step is to log constraints for breaking symmetries. The first symmetry considered is the symmetry

$$\pi := (p_{11}p_{43})(p_{12}p_{42})(p_{13}p_{41})(p_{21}p_{23})(p_{31}p_{33}),$$

which is the symmetry that swaps pigeons 1 and 4, and simultaneously swaps holes 1 and 3. In this work, we just take the set of symmetries to break on for granted and we will not elaborate on the possible reasons why this peculiar symmetry was chosen. As explained in our section on symmetry breaking, in order to break

this symmetry, first an exponential encoding of a lex-leader constraint is added using the dominance rule, as follows

```
L27  dom -1 p43 1 p11 -2 p42 2 p12 -4 p41 4 p13 -8 p33 8 p31 -32 p31 32 p33 -64 p13 64 p41
       ↪  -128 p12 128 p42 -256 p11 256 p43 -512 p23 512 p21 -2048 p21 2048 p23  >= 0 ;
       ↪   p11 -> p43 p12 -> p42 p13 -> p41 p21 -> p23 p23 -> p21 p31 -> p33 p33 -> p31
       ↪  p41 -> p13 p42 -> p12 p43 -> p11 ;  begin
L28    proofgoal #2
L29      p -1 -2 +
L30      c -1
L31    qed
```

These instruction tell *VeriPB* to use the dominance rule to derive (and add to $\mathscr{D}$) the constraint in Line 27[4], which expresses that the assignment in question is lexicographically smaller than it's symmetric counterpart. As expected, the variables related to pigeon 2 occur with the highest coefficients (since when instantiating the order, they were given the highest priority).

The actual instruction for *VeriPB* does not just contain the constraint to be derived by dominance, but also specifies

- The witness, which in this case is just the symmetry, in Line 27, and
- A subproof of one of the proof obligations in Lines 28–31.

As far as the subproof is concerned: to apply the dominance rule, we need to show that the two implications

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{C}\restriction_\omega \cup \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z}) \cup \{f\restriction_\omega \ \leq f\} \tag{27a}$$

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega) \vdash \bot \tag{27b}$$

hold. To do so, *VeriPB* generates the following proof obligations:

1. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \mathscr{O}_\preceq(\vec{z}\restriction_\omega, \vec{z})$
2. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \neg\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$
3. $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash \{f\restriction_\omega \ \leq f\}$

4–25 $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash B$ for each $B \in \mathscr{C}\restriction_\omega$.

Except for the second proof obligation, all of them can be proved automatically, for instance since we are working in the context of a decision problem where $f = 0$, the third one is trivial. Since ω is a syntactic symmetry of $\mathscr{C}$ (which is at this point still equal to the input), also the last ones are trivial. The proof of this second proof obligation goes as follows. First, *VeriPB* (automatically) adds the constraint $\neg C$, which (after simplification) equals:

$$255p_{11} + 126p_{12} + 60p_{13} + 1536p_{21} + 1536\overline{p_{23}} + 24p_{31} + 24\overline{p_{33}} + 60\overline{p_{41}} + 126\overline{p_{42}} + 255\overline{p_{43}} \geq 2002 \quad \text{(C23)}$$

Next, *VeriPB* (again, automatically) adds the constraint $\mathscr{O}_\preceq(\vec{z}, \vec{z}\restriction_\omega)$, which, after simplification, equals

$$255\overline{p_{11}} + 126\overline{p_{12}} + 60\overline{p_{13}} + 1536\overline{p_{21}} + 1536p_{23} + 24\overline{p_{31}} + 24p_{33} + 60p_{41} + 126p_{42} + 255p_{43} \geq 2001 \quad \text{(C24)}$$

Now the instruction 29 simply states that the last added constraint (i.e., Equation (C24)) and the one added before Equation (C23)) should be added resulting in

$$255 + 126 + 60 + 1536 + 1536 + 24 + 24 + 60 + 126 + 255 \geq 2002 + 2001, \tag{C25}$$

[4]Notice that this constraint contains some duplicate variables, because of being generated automatically; later on this will be simplified.

which is a contradiction. The line 30 states that the this is indeed a contradiction and the subproof for this proof obligation is ended. Finally, when this proof is finished and all other proof obligation have been automatically checked, the new constraint

$$\begin{aligned}-p_{43} + 1p_{11} - 2p_{42} + 2p_{12} - 4p_{41} + 4p_{13} - 8p_{33} + 8p_{31} - 32p_{31} + 32p_{33} - 64p_{13} + 64p_{41} \\ -128p_{12} + 128p_{42} - 256p_{11} + 256p_{43} - 512p_{23} + 512p_{21} - 2048p_{21} + 2048p_{23} \geq 0\end{aligned} \tag{C26}$$

is added to $\mathscr{D}$. Afterwards, constraints (C23), (C24), and (C25) are removed, since they are constraints that were only valid for the subproof.

This constraint, by itself, is a lex-leader constraint for the symmetry at hand. However, since we are in the context of SAT solving, it still has to be translated to a set of clauses, which is what happens next. First, (12a) is added with the redundance rule with the instruction

```
L32  red 1 y0  >= 1 ; y0 -> 1
```

which contains both the constraint

$$y_0 \geq 1 \tag{C27}$$

and the witness $y_0 \mapsto 1$ to apply the redundance rule. All proof obligations are checked automatically by *VeriPB*.

In our chosen lexicographic order, the most prominent variable is p_{21}. As such, the first clause for symmetry breaking is

$$\overline{p_{21}} \vee \pi(p_{21}) \doteq \overline{p_{21}} \vee p_{23},$$

which is a simplification of (12b), omitting the trivially true y_0. The constraint (C26) implies the above constraint (e.g., using weakening out all other variables in that constraint). Instead of giving the actual derivation, we can simply add it with the reverse unit propagation rule and let *VeriPB* figure out the details by

```
L33  u 1 ~p21 1 p23 >= 1 ;
```

resulting in

$$\overline{p_{21}} + p_{23} \geq 1. \tag{C28}$$

Next, the Tseitin variable y_1 is introduced with four redundance rule applications

```
L34  red 1 p23 1 ~y0 1 y1  >= 1 ; y1 -> 1
L35  red 1 ~p21 1 ~y0 1 y1  >= 1 ; y1 -> 1
L36  red 1 ~y1 1 y0  >= 1 ; y1 -> 0
L37  red 1 ~y1 1 ~p23 1 p21  >= 1 ; y1 -> 0
```

each of them also mentioning the witness mapping y_1 either to 0 or to 1, resulting in the constraints

$$p_{23} + \overline{y_0} + y_1 \geq 1 \tag{C29}$$

$$\overline{p_{21}} + \overline{y_0} + y_1 \geq 1 \tag{C30}$$

$$\overline{y_1} + y_0 \geq 1, \text{and} \tag{C31}$$

$$\overline{y_1} + \overline{p_{23}} + p_{21} \geq 1 \tag{C32}$$

corresponding to the constraints (12c)–(12f).

Before repeating this procedure for the next variable, we use the recently derived constrain to cancel out the dominant terms in constraint (C26) with the instructions

```
L38  p 26 32 2048 * +
L39  d 26
```
The first of these instructions results in adding (C32) 2048 times to (C26), resulting in

$$255\overline{p_{11}}+126\overline{p_{12}}+60\overline{p_{13}}+512p_{21}+512\overline{p_{23}}+24\overline{p_{31}}+24p_{33}+60p_{41}+126p_{42}+255p_{43}+2048\overline{y_1}\geq 977. \tag{C33}$$

The last of these instructions deletes (C26) from $\mathscr{D}$ since it will no longer be required.

After p_{21}, the next most important variable is p_{22}. However, since our symmetry π at hand maps p_{22} to itself, no symmetry breaking clauses are added for it. The next variable in the ordering is p_{23}, which is mapped to p_{21}, resulting in the (conditional on y_1) symmetry breaking constraint

$$\overline{y_1}+\overline{p_{23}}+p_{21}\geq 1 \tag{C34}$$

obtained by the instruction
```
L40  u 1 ~y1 1 ~p23 1 p21 >= 1 ;
```
Next, as before, the next Tseitin variable y_2 is introduced with the redundance rule using
```
L41  red 1 p21 1 ~y1 1 y2  >= 1 ; y2 -> 1
L42  red 1 ~p23 1 ~y1 1 y2  >= 1 ; y2 -> 1
L43  red 1 ~y2 1 y1  >= 1 ; y2 -> 0
L44  red 1 ~y2 1 ~p21 1 p23  >= 1 ; y2 -> 0
```
resulting in the constraints

$$p_{21}+\overline{y_1}+y_2\geq 1 \tag{C35}$$

$$\overline{p_{23}}+\overline{y_1}+y_2\geq 1 \tag{C36}$$

$$y_1+\overline{y_2}\geq 1, \text{ and} \tag{C37}$$

$$\overline{p_{21}}+p_{23}+\overline{y_2}\geq 1 \tag{C38}$$

As before, our pseudo-Boolean symmetry breaking constraint is simplified with
```
L45  p 33 38 512 * +
L46  d 33
```
where the first instruction again cancels out the dominant terms (replacing them by y-variables) to express a conditional symmetry breaking constraint, resulting in

$$255\overline{p_{11}}+126\overline{p_{12}}+60\overline{p_{13}}+24\overline{p_{31}}+24p_{33}+60p_{41}+126p_{42}+255p_{43}+2048\overline{y_1}+512\overline{y_2}\geq 465 \tag{C39}$$

The next most important variable in our chosen order is p_{11}, which is mapped to p_{43}, resulting in the symmetry breaking constraint

$$\overline{p_{11}}+p_{43}+\overline{y_2}\geq 1 \tag{C40}$$

added by
```
L47  u 1 ~y2 1 ~p11 1 p43 >= 1 ;
```
To see that this constraint indeed follows by reverse unit propagation, we observe that whenever y_2 holds, so do y_1 and y_0 (by (C37) and (C31). If furthermore $\overline{p_{43}}$ and p_{11} hold, (C39) simplifies to

$$126\overline{p_{12}}+60\overline{p_{13}}+24\overline{p_{31}}+24p_{33}+60p_{41}+126p_{42}\geq 465,$$

which can never be satisifed since the coefficients on the left add up to 420.

The process of introducing a new Tseitin variable is the same as before, resulting in the addition of the following constraints to $\mathscr{D}$:

$$p_{43} + \overline{y_2} + y_3 \geq 1 \tag{C41}$$

$$\overline{p_{11}} + \overline{y_2} + y_3 \geq 1 \tag{C42}$$

$$y_2 + \overline{y_3} \geq 1 \tag{C43}$$

$$p_{11} + \overline{p_{43}} + \overline{y_3} \geq 1 \tag{C44}$$

This last constraint can then again be used to simplify (C39) with

```
L48  p 39 44 256 * +
```

resulting in

$$p_{11} + 126\overline{p_{12}} + 60\overline{p_{13}} + 24\overline{p_{31}} + 24p_{33} + 60p_{41} + 126p_{42} + \overline{p_{43}} + 2048\overline{y_1} + 512\overline{y_2} + 256\overline{y_3} \geq 211 \tag{C45}$$

This process continues by considering all variables not stabilized by π in the order used for lexicographic ordering.

Logging the breaking of more symmetries

Afterwards, more symmetries are broken, and the resulting clauses logged. The process is completely the same as with the first symmetry. In our example, *BreakID* decided to break the following symmetries next:

$$(p_{11}p_{12})(p_{21}p_{32})(p_{22}p_{31})(p_{23}p_{33})(p_{41}p_{42})$$
$$(p_{21}p_{11})(p_{22}p_{12})(p_{23}p_{13})$$
$$(p_{11}p_{31})(p_{12}p_{32})(p_{13}p_{33})$$
$$(p_{31}p_{41})(p_{32}p_{42})(p_{33}p_{43})$$
$$(p_{21}p_{22})(p_{11}p_{12})(p_{31}p_{32})(p_{41}p_{42}), \text{ and}$$
$$(p_{22}p_{23})(p_{12}p_{13})(p_{32}p_{33})(p_{42}p_{43}).$$

The first of these symmetries swaps holes 1 and 2 and simultaneously swaps pigeons 2 and 3. It is important to note here that the breaking of these six symmetries by no means interacts with the previously derived breaking clauses: the order remains unchanged and all previously added constraints were added to $\mathscr{D}$, hence $\mathscr{C}$ still consists only of the input formula.

D Technical Appendix on Proof Logging for CP Symmetry Breaking

In the "Symmetries in Constraint Programming" subsection we describe how we can use proof logging in a constraint programming setting to verify an algorithm for solving the Crystal Maze puzzle. In this appendix we describe the key ideas behind an implementation of this algorithm; source code to run the demo is located in the `tools/crystal-maze-solver` directory of the code and data repository[5], and full instructions are given in the `tools/crystal-maze-solver/README.md` file. An outline of this work follows below.

[5] See https://doi.org/10.5281/zenodo.6373986.

We modelled the Crystal Maze puzzle as a constraint satisfaction problem in the natural way: we have a decision variable for each circle, whose values are the possible numbers that can be taken, and an all-different constraint over all decision variables. We use a table constraint for each edge, for simplicity. We also included symmetry elimination constraints.

We implemented this model inside a small proof-of-concept CP solver `src/crystal_maze.cc` that we created for this paper. (Full proof logging for CP is an entire research program in its own right, which we do not claim to have carried out—what we do claim, though, is that our contribution shows that symmetries do not stand in the way of this work.) When executed, the solver compiles this high level CP model to a pseudo-Boolean model, which it will output as `crystal_maze.opb`. This is done following the framework introduced by Elffers et al. [2020], but as well as using a one-hot (direct) encoding of CP decision variables to PB variables, it additionally creates channelled greater-or-equal PB variables for each CP variable-value. Note that the encoding of the table constraints also introduces additional auxiliary variables.

Then, as it solves the problem, the solver outputs `crystal_maze.veripb`, which provides a proof that it has found all non-symmetric solutions. (Note that our solver maintains generalised arc consistency on the all-different and table constraints, and so is performing propagation that requires explicit justification in the proof log.) These two outputs can be verified using *VeriPB*.

To verify that the symmetry constraints introduced in the high level model are actually valid, we can remove them from the pseudo-Boolean model and introduce them as part of the proof instead. We describe how to do this editing in `README.md`. We also include a script `make-symmetries.py` that will output the necessary proof fragment to reintroduce the symmetry constraints. The output of this script can be verified on top of the reduced pseudo-Boolean model using our modified version of *VeriPB*, with or without the remainder of the proof—that is, we can both verify that the constraints introduced are valid (in that they do not alter the satisfiability of the model), and that they line up with the actual execution of the solver.

E Technical Appendix on Proof Logging for Vertex Dominance in Max-Clique Solving

Throughout this last appendix we let $G = (V, E)$ denote an undirected, unweighted graph without self-loops with vertices V and edges E. We write $N(u)$ to denote the *neighbours* of a vertex $u \in V$, i.e., the set of vertices $N(u) = \{w \mid (u, w) \in E\}$ that are adjacent to u in the graph, and define neighbours of sets of vertices in the natural way by taking unions $N(U) = \bigcup_{u \in U} N(u)$.

We say that u *dominates* v if

$$N(u) \setminus \{v\} \supseteq N(v) \setminus \{u\} \tag{28}$$

holds, which intuitively says that the neighbourhood of u is at least as large as that of v. It is straightforward to verify that this domination relation is transitive.

When representing the maximum clique problem in pseudo-Boolean form, we overload notation and identify every vertex $v \in V$ with a Boolean variable, where $v = 1$ means that the vertex v is in the clique. The task is to maximize $\sum_{v \in V} v$ under the constraints that all chosen vertices should be neighbours, but since, syntactically speaking, we require an objective function to be minimized, we obtain

$$\min \textstyle\sum_{v \in V} \overline{v} \tag{29a}$$

$$\overline{v} + \overline{w} \geq 1 \qquad [\text{for all } (v, w) \notin E] \tag{29b}$$

as the formal pseudo-Boolean specification of the problem.

Algorithm 1: Max clique algorithm without dominance.

```
1  MaxCliqueSearch(G, V_rem, C_curr, C_best) :
2  E_rem ← E(G) ∩ (V_rem × V_rem);
3  G_rem ← (V_rem, E_rem);
4  if |C_curr| > |C_best| then
5      C_best ← C_curr;
6  (S_1, ..., S_m) ← colour classes in colouring of G_rem ;
7  j ← m;
8  while j ≥ 1 and |C_curr| + j > |C_best| do
9      for v ∈ S_j do
10         C_best ← MaxCliqueSearch(G, V_rem ∩ N(v), C_curr ∪ {v}, C_best);
11     V_rem ← V_rem \ S_j;
12     j ← j − 1;
13 return C_best;
```

High-Level Description of the Max Clique Solver

At a high level, the maximum clique solver of McCreesh and Prosser [2016], but before addition of vertex dominance breaking, works as described in Algorithm 1. The first call to the MaxCliqueSearch algorithm is with parameters G, $V_{\mathrm{rem}} = V$, $C_{\mathrm{curr}} = \emptyset$, and $C_{\mathrm{best}} = \emptyset$.

When MaxCliqueSearch is called with a candidate clique C_{curr}, the best solution so far C_{best}, and a subset of vertices V_{rem}, it considers the residual graph $G_{\mathrm{rem}} = (V_{\mathrm{rem}}, E_{\mathrm{rem}})$ assumed to be defined on all vertices in $V \setminus C_{\mathrm{curr}}$ that are neighbours of all $c \in C_{\mathrm{curr}}$. Thus, the set V_{rem} contains all vertices to which C_{curr} could possibly be extended. The algorithm produces a colouring of G_{rem}, which we assume results in m disjoint colour classes $(S_1, \ldots, S_m)$ such that $V_{\mathrm{rem}} = \bigcup_{i=1}^{m} S_i$. It is clear that any clique extending C_{curr} can contain at most one vertex from every colour class S_i. The clique search algorithm now iterates over all colour classes in the order $S_m, S_{m-1}, \ldots, S_1$. Whenever the clique is extended with a new vertex, a new recursive call to MaxCliqueSearch is made. Therefore, when we reach S_j in the loop, we are considering the case when all vertices in $S_m, S_{m-1}, \ldots, S_{j+1}$ have been rejected. For this reason, if the condition $|C_{\mathrm{curr}}| + j > |C_{\mathrm{best}}|$ fails to hold, we know that the current clique candidate cannot possibly be extended to a clique that is larger than what we have already found in C_{best}. At the end of the first call $\mathsf{MaxCliqueSearch}(G, V, C_{\mathrm{curr}} = \emptyset, C_{\mathrm{best}} = \emptyset)$, after completion of all recursive subcalls, the vertex set C_{best} will be a clique of maximum size in G. A certifying version of essentially this algorithm with *VeriPB* proof logging was presented by Gocht et al. [2020a]. It might be worth noting in this context that one quite interesting challenge is to justify the backtracking performed when the condition $|C_{\mathrm{curr}}| + j > |C_{\mathrm{best}}|$ fails, and this is one place where the strength of the pseudo-Boolean reasoning in the cutting planes proof system is very helpful (as opposed to the clausal reasoning in, e.g., *DRAT*).

The vertex dominance breaking of McCreesh and Prosser [2016] is based on the following observation: If the algorithm is about to consider $v \in S_j$ in the innermost for loop on line 9 in Algorithm 1, but has previously considered a vertex $u \in \bigcup_{i=j}^{m} S_i$ that dominates v in the sense of (28), then it is safe to ignore v. If the algorithm would find a solution that includes v but not u, then we can swap u for v and obtain a solution that is at least as good.

In pseudo-Boolean notation, this would correspond to adding the constraint $u + \overline{v} \geq 1$ to the formula, but there is no way this can be semantically derived from the constraints (29a) or the requirement to mini-

mize (29b). Therefore, the proof logging method in [Gocht et al., 2020a] is inherently unable to deal with such constraints.

In general, the vertex dominance breaking as described above does not need to break ties consistently. By this we mean that if u and v dominate each other, in principle it might happen that in a given branch of the search tree, u is chosen to dominate v, while in another one, v is chosen to dominate u, simply because of the order in which nodes are considered. While in principle, our proof logging methods can be adapted to work in this case, the argument is subtle. Luckily, it turns out that in practical implementations, tie breaking only happens in a consistent manner.

Fact 13. *In the vertex dominance breaking of McCreesh and Prosser [2016], there exists a total order $\succ_G$ on the set V of vertices such that whenever v is ignored because u has previously been considered, $u \succ_G v$.*

Moreover, this order $\succ_G$ is known before the algorithm starts: $u \succ_G v$ holds if u has a larger degree than v, or in case they have the same degree but the identifier used to represent u internally is larger than that of v. To see that this order indeed guarantees consistent tie breaking, we provide some properties of the actual implementation of the algorithm.

1. If u and v dominate each other and are not adjacent, u and v are guaranteed to be in the same coloring class. If furthermore $u \succ_G v$, u is considered before v in the loop in Line 8 (due to the order in which this for loop iterates over the nodes).

2. If u and v dominate each other, *are* adjacent, and $u \succ_G v$, then u is assigned a *larger* coloring class than v (due to the order in which the (greedy) coloring algorithm in Line 6 iterates over the nodes). Hence, also in this case u will be considered before v.

In what follows below, we will explain

- first, how the redundance rule introduced to *VeriPB* by Gocht and Nordström [2021] could in principle be used to provide proof logging for vertex dominance breaking, but that this seems hard to get to work in practice; and

- then, how the dominance rule introduced in this paper can be used to resolve the practical problems in a very simple way.

An implementation for both techniques can be found in the code and data repository.[5]

Vertex Dominance with the Redundance Rule

In order to provide proof logging for vertex dominance breaking using the redundance rule, we could in theory proceed as follows. First, we let the solver check the vertex dominance condition (28) for all pairs of vertices u, v in V.

Before starting the solver, we add all pseudo-Boolean constraints for vertex dominance breaking using the redundance rule. For all u, v such that u dominates v and $u \succ_G v$, we derive the *vertex dominance breaking constraint*

$$u + \overline{v} \geq 1 \ , \tag{30}$$

doing so *in decreasing order* for u with respect to $\succ_G$. Our witness for the redundance rule derivation of (30) will be $\omega = \{u \mapsto v, v \mapsto u\}$, i.e., ω will simply swap the dominating and dominated vertices. Hence, the objective function (29a) is syntactically unchanged after substitution by ω, and so the condition in (5) that the objective should not increase is always vacuously satisfied.

We need to argue that deriving the vertex dominance breaking constraints (30) is valid in our proof system. Towards this end, suppose we are in the middle of the process of adding such constraints and are currently considering $u + \overline{v} \geq 1$ for u dominating v and $u \succ_G v$. Let $\mathscr{C} \cup \mathscr{D}$ be the set of constraints in the current configuration. In order to add $u + \overline{v} \geq 1$, we need to show that

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg(u + \overline{v} \geq 1)\} \tag{31a}$$

can be used to derive all constraints in

$$(\mathscr{C} \cup \mathscr{D} \cup \{u + \overline{v} \geq 1\})\restriction_\omega \tag{31b}$$

by the cutting planes method (i.e., without any extension rules).

Starting with the vertex dominance constraint being added, note that from the negated constraint $\neg(u + \overline{v} \geq 1) \doteq \overline{u} + v \geq 2$ in (31a) we immediately obtain

$$\overline{u} \geq 1 \tag{32a}$$

$$v \geq 1 \tag{32b}$$

as RUP constraints, meaning that the weaker constraint $(u + \overline{v} \geq 1)\restriction_\omega \doteq v + \overline{u} \geq 1$ is also RUP with respect to the constraints in (31a).

Consider next any non-edge constraints $\overline{x} + \overline{y} \geq 1$ in (29b) in the original formula. Clearly, such constraints are only affected by ω if $\{u, v\} \cap \{x, y\} \neq \emptyset$; otherwise they are present in both (31a) and (31b) and there is nothing to prove. Any non-edge constraint $\overline{v} + \overline{y} \geq 1$ containing $\overline{v}$ will after application of ω contain $\overline{u}$, and will hence be RUP with respect to (32a) and hence also with respect to (31a). For non-edge constraints $\overline{u} + \overline{y} \geq 1$ with $y \neq v$, substitution by ω yields $(\overline{u} + \overline{y} \geq 1)\restriction_\omega \doteq \overline{v} + \overline{y} \geq 1$. Since by assumption u dominates v and $y \neq v$ is not a neighbour of u, it follows from (28) that y is not a neighbour of v either. Hence, the input formula in (31a) already contains the desired non-edge constraint $\overline{v} + \overline{y} \geq 1$.

It remains to analyse what happens to vertex dominance breaking constraints

$$x + \overline{y} \geq 1 \tag{33}$$

that have already been added to $\mathscr{D}$ before the dominance breaking constraint $u + \overline{v} \geq 1$ that we are considering now. Again, such a constraint is only affected by ω if $\{u, v\} \cap \{x, y\} \neq \emptyset$; otherwise it is present in both (31a) and (31b). We obtain the following case analysis.

1. $x = u$: In this case, $(x + \overline{y} \geq 1)\restriction_\omega \doteq v + \overline{\omega(y)} \geq 1$, which is RUP with respect to $v \geq 1$ in (32b) and hence also with respect to (31a).

2. $x = v$: This is impossible, since $u \succ_G v$ and any dominance breaking constraints with $v = x$ as the dominating vertex will be added only once we are done with u as per the description right below (30).

3. $y = u$: In this case x dominates u. Since $x \succ_G u$, $u \succ_G v$, and u dominates v, by transitivity we have $x \succ_G v$ and also that x dominates v. Hence, the breaking constraint $x + \overline{v} \geq 1$ has already been added to $\mathscr{D}$. But since $u \neq x \neq v$, we see that our desired constraint is $(x + \overline{u} \geq 1)\restriction_\omega \doteq x + \overline{v} \geq 1$, which is precisely this previously added constraint.

4. $y = v$: Here we see that the desired constraint $(x + \overline{y} \geq 1)\restriction_\omega \doteq \omega(x) + \overline{u} \geq 1$ is again RUP with respect to (31a).

This concludes our proof that all vertex dominance breaking constraints that are consistent with our constructed linear order $\succ_G$ can be added and certified by the redundance rule before the solvers starts searching for cliques.

So all of this works perfectly fine in theory. The problem that rules out this approach in practice, however, is that the solver will not have the time to compute the dominance relation between vertices in advance, since this is far too costly and does not pay off in general. Instead the solver designed by McCreesh and Prosser [2016] will detect and apply vertex dominance relations on the fly during search. And from a proof logging perspective this is too late—during search, when $\mathscr{C} \cup \mathscr{D}$ will also contain constraints justifying any backtracking made, our proof logging approach above no longer works. The constraints added to the proof log to justify backtracking are no longer possible to derive when substituted by ω as in (31b), for the simple reason that they are not semantically implied by (31a). One possible way around this would be to run the solver twice—the first time to collect all information about what vertex dominance breaking will be applied, and then the second time to do the actual proof logging—but this seems like quite a cumbersome approach. Moreover, even when doing so, our argument of correctness uses that the set of pairs (u, v) for which a constraint $u + \overline{v} \geq 1$ is derived is transitively closed, i.e., we would still add more constraints than what the solver actually needs.

We deliberately discuss this problem in some detail here, because this is an example of an important and nontrivial challenge that shows up also in other settings when designing proof logging for other algorithms. It is not sufficient to just come up with a proof logging system that is strong enough in principle to certify the solver reasoning (which the redundance rule is for the clique solver with vertex dominance breaking, as shown above). It is also crucial that the solver have enough information available at the right time and can extract this information efficiently enough to actually be able to emit the required proof logging commands with low enough overhead. For constraint programming solvers, it is not seldom the case that the solver knows for sure that some variable should propagate to a value, because the domain has shrunk to a singleton, or that the search should backtrack because some variable domain is empty, but that the solver cannot reconstruct the detailed derivation steps required to certify this without incurring a massive overhead in running time (e.g., since the reasoning has been performed with bit-parallel logical operations). It is precisely for this reason that it is important that our proof system allow adding reverse unit propagation (RUP) constraints. This makes it possible for the solver to claim facts that it knows to be true, and that it knows can be easily verified, while leaving the work of actually producing a detailed justification to the proof checker.

Vertex Dominance with the Dominance Rule

Similar to the case of the redundance rule we will make use of Fact 13. Before staring the proof logging, we use the order change rule to activate the lexicographic order on the the assignments to the vertices/variables induced by $\succ_G$.

Suppose now that the solver is running and that the current candidate clique is C_{curr}. The solver has an ordered list of unassigned candidate vertices that it is iterating over when considering how to enlarge this clique, and this list is defined by the colour classes $(S_m, S_{m-1}, \ldots, S_1)$. (We note that this ordered list depends on C_{curr}, and would be different for a different clique C'_{curr}.) Suppose the next vertex in that list is v. Then when it is time to make the next decision on line 9 in Algorithm 1 about enlarging the clique, we can apply the following decision algorithm:

1. If there exists a vertex u that has already been considered in the current iteration and that dominates v (and hence for which $u \succ_G v$), then discard v by vertex dominance and add the constraint $u + \overline{v} \geq 1$ by the dominance rule with witness $\omega = \{u \mapsto v, v \mapsto u\}$. We will below explain in detail why this is possible.

2. Otherwise, enlarge C_{curr} with v and make a recursive call.

When the solver has explored all ways of enlarging C_{curr} and is about to backtrack, here is what will happen on the proof logging side (where we refer to [Gocht et al., 2020a] for a more detailed description of how proof logging for backtracking CP solvers works in general):

1. For every u that was explored in an enlarged clique $C_{\mathrm{curr}} \cup \{u\}$, when backtracking the solver will already have added $\overline{u} + \sum_{w \in C_{\mathrm{curr}}} \overline{w} \geq 1$ as a RUP constraint.

2. The solver now inserts the explicit cutting planes derivation required to justify that $|C_{\mathrm{curr}}| + j > |C_{\mathrm{best}}|$ must hold.

3. After this, the solver adds the claim that $\sum_{w \in C_{\mathrm{curr}}} \overline{w} \geq 1$ is a RUP constraint.

We need to argue why $\sum_{w \in C_{\mathrm{curr}}} \overline{w} \geq 1$ will be accepted as a RUP constraint, allowing the solver to backtrack. The RUP check for $\sum_{w \in C_{\mathrm{curr}}} \overline{w} \geq 1$ propagates $w = 1$ for all $w \in C_{\mathrm{curr}}$. This in turn propagates $u = 0$ for all explored vertices u by the backtracking constraints for $C_{\mathrm{curr}} \cup \{u\}$ added in step 1. The vertex dominance breaking constraints then propagate $v = 0$ for all vertices v discarded because of vertex domination. At this point, the proof checker has the same information that the solver had when it detected that the colouring constraint forced backtracking. This means that the proof checker will unit propagate to contradiction, and so the backtracking constraint $\sum_{w \in C_{\mathrm{curr}}} \overline{w} \geq 1$ is accepted as a RUP constraint.

We still need to explain how and why the pseudo-Boolean dominance rule applications allow deriving the constraint $u + \overline{v} \geq 1$ in case u dominates v (and hence $u \succ_G v$). Recall that the order used in our proof is the lexicographic order in duced by $\succ_G$. This means that if vertices/variables u and v are assigned by α in such a way as to violate a dominance breaking constraint $u + \overline{v} \geq 1$, then $\alpha \circ \omega$ will flip u to 1 and v to 0 to produce a lexicographically smaller assignment (since v is considered before u in the lexicographic order).

The conditions for the dominance rule are that we have to exhibit proofs of (7a) and (7b). In this discussion, let us focus on (7a) which says that starting with the constraints

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg(u + \overline{v} \geq 1)\} \tag{34a}$$

and using only cutting planes rules, we should be able to derive

$$\mathscr{C}\restriction_{\omega} \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z}) \cup \{f\restriction_{\omega} \ \leq f\} \ . \tag{34b}$$

Note first that our lexicographic order in fact does *not* in itself respect the objective function (29b). However, since ω just swaps two variables it leaves the objective syntactically unchanged, meaning that the inequality $f\restriction_{\omega} \ \leq f$ in (7a) is seen to be trivially true.

As in our analysis of the redundance rule, from (34a) we obtain $\overline{u} \geq 1$ and $v \geq 1$ as in (32a)–(32b), and $\mathscr{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z})$ is easily verified to be RUP with respect to these constraints, since what the formula says after cancellation is precisely that $v \geq u$.

It remains to consider the pseudo-Boolean constraints in the solver constraint database $\mathscr{C} \cup \mathscr{D}$. The crucial difference from the redundance rule is that we no longer have to worry about proving $\mathscr{D}\restriction_{\omega}$ in (34b)—we only need to show how to derive $\mathscr{C}\restriction_{\omega}$. But this means that all we need to consider are the non-edge constraints in (29b), and we already explained in our analysis for the redundance rule derivation that the fact that u dominates v means that for any non-edge constraints affected by ω their substituted versions are already there as input constraints or are easily seen to be RUP constraints. In addition to these non-edge constraints there might also be all kinds of interesting derived constraints in $\mathscr{D}$, but the dominance rule says that we can ignore those constraints.

Finally, although we skip the details here, it is not hard to argue analogously to what has been done above to show that $\neg C \doteq \neg(u + \overline{v} \geq 1)$ and $\mathscr{O}_{\preceq}(\vec{z}, \vec{z}\restriction_{\omega})$ in (7b) together unit propagate to contradiction. This concludes our discussion of how to certify vertex dominance breaking in the maximum clique solver by McCreesh and Prosser [2016] using the pseudo-Boolean dominance rule introduced in this paper.

Paper C

Cutting to the Core of Pseudo-Boolean Optimization: Combining Core-Guided Search with Cutting Planes Reasoning

Jo Devriendt,[1,2,3] **Stephan Gocht,**[1,2] **Emir Demirović,**[4] **Jakob Nordström,**[2,1] **Peter J. Stuckey**[5]
[1] Lund University, Lund, Sweden
[2] University of Copenhagen, Copenhagen, Denmark
[3] KU Leuven, Leuven, Belgium
[4] TU Delft, Delft, The Netherlands
[5] Monash University, Melbourne, Australia
jo.devriendt@cs.lth.se, stephan.gocht@cs.lth.se, e.demirovic@tudelft.nl, jn@di.ku.dk, peter.stuckey@monash.edu

Abstract

Core-guided techniques have revolutionized Boolean satisfiability approaches to optimization problems (MaxSAT), but the process at the heart of these methods, strengthening bounds on solutions by repeatedly adding cardinality constraints, remains a bottleneck. Cardinality constraints require significant work to be re-encoded to SAT, and SAT solvers are notoriously weak at cardinality reasoning. In this work, we lift core-guided search to pseudo-Boolean (PB) solvers, which deal with more general PB optimization problems and operate natively with cardinality constraints. The cutting planes method used in such solvers allows us to derive stronger cardinality constraints, which yield better updates to solution bounds, and the increased efficiency of objective function reformulation also makes it feasible to switch repeatedly between lower-bounding and upper-bounding search. A thorough evaluation on applied and crafted benchmarks shows that our core-guided PB solver significantly improves on the state of the art in pseudo-Boolean optimization.

Introduction

The *Boolean satisfiability (SAT) problem* plays a fascinating dual role in computer science. Although it is an archetypal hard problem—proven *NP*-complete in (Cook 1971; Levin 1973) and widely believed to be exponentially hard in theory—at the same time it serves as the modelling language for the *conflict-driven clause learning (CDCL)* SAT solvers (Bayardo Jr. and Schrag 1997; Marques-Silva and Sakallah 1999; Moskewicz et al. 2001) that have emerged over the last two decades as highly practical tools for solving large-scale real-world problems in a wide range of application areas (Biere et al. 2021). This success has also led to exports of the conflict-driven paradigm beyond SAT solving to, e.g., *SAT-based optimization (MaxSAT)* (Fu and Malik 2006), *pseudo-Boolean (PB) optimization* (Chai and Kuehlmann 2005; Sheini and Sakallah 2006), *constraint programming (CP)* (Ohrimenko, Stuckey, and Codish 2009; Stuckey 2010), and *mixed integer programming (MIP)* (Achterberg 2007).

Our starting point in this work is the MaxSAT problem, which differs from SAT in that some of the constraints are declared to be soft, but with associated penalties for violating them, and where the goal is to minimize the total penalty of violated constraints. The *core-guided* approach introduced by (Fu and Malik 2006) optimistically assumes that this penalty is zero, and then tries to solve the resulting SAT problem under this assumption as described in (Eén and Sörensson 2003). If this attempt fails, the solver returns a *core* explaining why the assumption was too good to be true, and such cores are repeatedly used to update the estimate of the optimal solution and make new attempts with revised assumptions. Such techniques play a crucial role for the performance of modern MaxSAT solvers (Morgado et al. 2013), and have also been adapted to other paradigms such as *answer set programming (ASP)* (Andres et al. 2012) and constraint programming (Gange et al. 2020).

A technical barrier for efficient implementations of core-guided search, however, is that the process of using cores to strengthen bounds requires dealing with cardinality constraints. Such constraints are cumbersome to encode in the low-level language of propositional logic, and the *resolution* method on which CDCL SAT solvers are based (Beame, Kautz, and Sabharwal 2004) has severe limitations when it comes to cardinality reasoning (Haken 1985), affecting even core-guided approaches that use clauses to explain propagations by the cardinality constraints (Manquinho, Marques-Silva, and Planes 2009; Alviano, Dodaro, and Ricca 2015).

Our Contribution The simple but crucial observation underlying our work is that the MaxSAT problem of minimizing a weighted sum of penalties subject to constraints expressed in conjunctive normal form (CNF) is just a special case of PB optimization (also known as 0-1 *integer linear programming*). An intriguing fact in this context is that there are PB solvers that borrow the conflict-driven paradigm from SAT but perform their reasoning using the *cutting planes* method (Cook, Coullard, and Turán 1987). Cardinality constraints are no problem for such solvers, since they operate with even more general PB constraints, and it is known that cutting planes applied to cardinality constraints is exponentially more powerful than resolution.

In view of this, it might seem like an attractive, and even obvious, proposition to combine PB reasoning with

core-guided search. In practice, however, harnessing the theoretical power of the cutting planes method in PB solvers has turned out to be very challenging, to the extent that the best PB optimization solver NAPS (Sakai and Nabeshima 2015) in the most recent PB Competition in 2016 (`www.cril.univ-artois.fr/PB16/`) instead rewrites the input to CNF and runs a CDCL solver. One of the problems with PB solvers is that the increased degree of freedom make it hard to know how to best explore the search space, and for the same reason it is not a priori obvious what would be "the right way" of generalizing core-guided techniques to a PB setting.

In this paper, we report on our work on designing algorithms and heuristics for core-guided PB solving. We implement different approaches in the state-of-the-art PB solver ROUNDINGSAT (Elffers and Nordström 2018), and perform an extensive evaluation on applied and crafted benchmarks from different domains.

The one-sentence summary of our results is that adding core-guided techniques dramatically improves the solver. Core-guided search with clausal cores, as in SAT, already enhances performance, but the cutting planes method also allows the solver to derive stronger, non-clausal, cores. These cores lead to better updates of the solution bounds, meaning that the solver can zoom in faster on the optimal solution. Even more strikingly, the fact that all cores and objective function reformulations can be expressed in the native format of the solver means that there is very little overhead. This makes it possible to go beyond *core-boosting* (Berg, Demirović, and Stuckey 2019), which combines an initial core-guided search phase with a longer upper-bounding linear search phase, and implement a fully *hybrid* mode that switches repeatedly back and forth between core-guided search and linear search at very little cost, similar to *interleaving* in ASP (Alviano et al. 2015). This hybrid mode is what gives the best performance overall.

We have also evaluated popular heuristics from core-guided MaxSAT solvers such as using *stratification* (Ansótegui et al. 2012) and *independent cores* (Berg and Järvisalo 2017) during core-guided search, and fixing the phase to that of the incumbent solution during linear search (Demirović, Chu, and Stuckey 2018; Demirović and Stuckey 2019) rather than using standard phase saving as in (Pipatsrisawat and Darwiche 2007). Here the results are not so clear-cut. How to set the phase does not seem to have a decisive influence. Stratification and independent cores have a much less positive impact than we expected—these settings are good for some classes of benchmarks, but for others they make performance notably worse (which is particularly pronounced for independent cores).

Overall, adding core-guided search to ROUNDINGSAT dramatically improves the solver, to the extent that it is now better by a wide margin than the latest versions of both NAPS (Sakai and Nabeshima 2015) and SAT4J (Le Berre and Parrain 2010) for the PB Competition 2016 benchmarks.

Preliminaries

We start with a review of the basics of PB solving—this material is standard, and can be found, e.g., in (Buss and Nordström 2021). A *literal* ℓ over a Boolean variable x is x itself or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true) and where we define $\overline{\overline{x}} = x$ for convenience. A *PB constraint* C is a 0-1 integer linear inequality

$$\textstyle\sum_i a_i \ell_i \geq B \ , \tag{1}$$

which without loss of generality we 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)* B are non-negative integers. A cardinality constraint is a PB constraint in normalized form where all coefficients are 1. We use equality $\sum_i a_i \ell_i = B$ as syntactic sugar for the pair of inequalities $\sum_i a_i \ell_i \geq B$ and $\sum_i - a_i \ell_i \geq -B$ (but rewritten in normalized form).

The *weakening* rule $\mathsf{weaken}(C, \ell_j) \equiv \sum_{i \neq j} a_i \ell_i \geq B - a_j$ removes a literal ℓ_j from the constraint by subtracting its coefficient from the right-hand side, and $\mathsf{weaken}(C, L)$ for a set of literals L performs this operation for all $\ell_j \in L$. The division rule $\mathsf{divide}(C, d) \equiv \sum \lceil a_i/d \rceil \ell_i \geq \lceil B/d \rceil$ divides all coefficients and the degree by $d \in \mathbb{N}^+$ and rounds up. The operation $\mathsf{round2card}(C)$ computes a cardinality constraint over the literals in C with the degree equal to the minimum number of literals that must be set to true in order to satisfy C.

A *PB formula* is a conjunction $F = \bigwedge_j C_j$ of PB constraints. Note that a *clause* $\ell_1 \vee \cdots \vee \ell_k$ is equivalent to the constraint $\ell_1 + \cdots + \ell_k \geq 1$, so formulas in *conjunctive normal form (CNF)* are special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from literals to $\{0, 1\}$, where we write $\rho(x) = \rho(\overline{x}) = *$ if x is not in the domain of ρ and define $\rho(\overline{x}) = 1 - \rho(x)$ otherwise. If ρ is partial, then it is also referred to as a *restriction*, and the *restricted* constraint $C|_\rho$ is obtained by substituting values for all assigned variables and adjusting the degree appropriately, i.e.,

$$C|_\rho = \textstyle\sum_{\rho(\ell_i)=*} a_i \ell_i \geq B - \sum_{\rho(\ell_i)=1} a_i \ . \tag{2}$$

For a PB formula $F = \bigwedge_j C_j$ we define $F|_\rho = \bigwedge_j C_j|_\rho$. The constraint C is *satisfied* by ρ if $\sum_{\rho(\ell_i)=1} a_i \geq B$, (or, equivalently, if the restricted constraint (2) has a non-positive degree and hence is trivial). A PB formula is satisfied by ρ if all constraints in it are, in which case it is *satisfiable* and ρ is a *solution*. If there is no satisfying assignment, the formula is *unsatisfiable*.

A constraint C is said to *unit propagate* the literal ℓ under ρ if $C|_\rho$ cannot be satisfied unless $\ell \mapsto 1$. During *unit propagation* on F under ρ, we extend ρ iteratively by any propagated literals $\ell \mapsto 1$ 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 referred to as a *conflict*, since ρ' *violates* the constraint C in this case, and ρ' is called a *conflicting* assignment.

A *PB optimization* problem consists of a PB formula F and an objective function $O \equiv \sum_{i=1}^{m} a_i \ell_i + c_o$. We will abuse notation slightly and write $(a, \ell) \in O$ to obtain coefficient-literal pairs from the objective. Given an assignment ρ we write $\rho(O)$ to denote the value of the objective

function under ρ. Without loss of generality we assume that all coefficients in the objective are positive and that we want to minimize the objective function. An *optimal solution* is a satisfying assignment for F with minimum objective value $\rho(O)$ among all solutions.

Let $\text{Vars}(F)$ ($\text{Lits}(F)$) denote the variables (literals) appearing in F and analogously for O. Given a PB optimization problem, a *fresh variable* is a variable that does not appear in the formula or the objective function.

The idea of *linear search*, a widely used approach for PB optimization, is to find a solution ρ to the formula F, after which the constraint $O \leq \rho(O) - 1$ (in normalized form) is added to F. This can be repeated until F turns unsatisfiable, at which point the solution last found is the optimal solution.

Core-guided MaxSAT Solving *Maximum satisfiability (MaxSAT)* can be viewed as a PB optimization problem with a CNF formula, but many MaxSAT solvers use not only linear search but also *core-guided* approaches that work as follows (expressed in PB notation).

Given an objective function O, we build a set of assumptions $A = \text{Lits}(O)$ and solve the formula $F \cup \{\overline{\ell_i} \mid \ell_i \in A\}$. There are two cases: either we find a solution (which must be optimal, since the objective value is zero under A) or the problem is unsatisfiable. In the latter case, the solver can be made to return a subset $\kappa \subseteq A$ of the assumption that force unsatisfiability. This subset κ, treated as a clause $\sum_{\ell \in \kappa} \ell \geq 1$, is called an *unsatisfiable core*, and is implied by F since one of the assumptions must be falsified in any solution. Core-guided methods then reformulate the problem to take this information into account, deducing that the objective value must be at least $a_{min} = \min_{\ell_i \in \kappa} a_i$. The *OLL method* (Andres et al. 2012; Morgado, Dodaro, and Marques-Silva 2014) introduces new Boolean variables z_j that represent the (lower bounds of the) sum $1 + \sum_j z_j = \sum_{\ell \in \kappa} \ell$, and essentially rewrites the objective to $O + a_{min}(1 + \sum_j z_j - \sum_{\ell \in \kappa} \ell)$. This is a new MaxSAT problem, for which we can repeat the procedure described above again. The whole process terminates when a solution is found, which is guaranteed to be optimal since it assigns zero to all literals in the rewritten objective function.

Overview of the Optimization Algorithm

The general idea of our PB optimization approach is shown in Algorithm 1, which uses an *incremental* PB solver. The interface to the solver is similar to that of an incremental SAT solver (Eén and Sörensson 2003) and has two methods, one for adding constraints and one for solving the problem. The solve method $solve(A)$ also takes a (potentially empty) set of literals A and returns either $sat(\rho)$ where ρ is a full assignment satisfying the added constraints and the assumptions A; or $unsat(C)$ where C is a PB constraint implied by the added constraints that is falsified by the assumptions. We call this constraint a *core*. Note that in contrast to the MaxSAT setting such a core can now be an arbitrary, non-clausal PB constraint.

In each iteration, Algorithm 1 will refine either the lower or upper bound. If a solution is found a constraint is added that only allows strictly better solutions. If a core is returned we will reformulate the objective function so that the best known lower bound is the constant part of the objective and so that all coefficients remain non-negative.

Algorithm 1 PB Optimization with Core Extraction.

```
1:  procedure OPTIMIZE(F, O)
2:      lb ← 0;   ub ← ∞;   O' ← O
3:      solver.add(F)
4:      while ub − lb > 0 do
5:          pick set of assumption literals A ⊆ Lits(O')
6:          result ← solver.solve({¬ℓ_i | ℓ_i ∈ A})
7:          if result ≡ sat(ρ) then
8:              ub ← ρ(O)
9:                      ▷ improves best solution by at least 1
10:             solver.add(O < ub)
11:         else            ▷ unsatisfiable under assumptions
12:             let result ≡ unsat(C)
13:             lb ← improveBound(O', C)
14:                     ▷ improves lower bound by at least 1
15:             E ← encoding for reformulation variables
16:             O' ← reformulate(O', E)
17:             solver.add(E)
18:     return ub
```

Example 1. To illustrate Algorithm 1, suppose we want to minimize $x_1 + x_2 + x_3 + x_4$ subject to $x_1 + x_2 + 2x_5 \geq 2$ and $x_3 + x_4 - 2x_5 \geq 0$. If we start by assuming all variables to false, the solver obtains the core $x_1 + x_2 + x_3 + x_4 \geq 2$ by adding the two constraints in the formula together. This core gives us a lower bound of 2. Next we introduce new variables z_1, z_2 that encode the value of the sum of the x_i variables by adding the constraint $z_1 + z_2 + 2 = x_1 + x_2 + x_3 + x_4$ to the solver. Note that this equality represents a reformulation $z_1 + z_2 + 2$ of the objective function, so we can now continue by minimizing this reformulated objective $z_1 + z_2 + 2$, which contains the just derived lower bound 2 as constant term. Suppose we perform the next iteration without assumptions and let us say the solver produces a solution with objective value 3. We add the constraint $x_1 + x_2 + x_3 + x_4 < 3$ and continue. Because the gap between best solution and lower bound is 1 the solver will terminate in the next iteration by finding an optimal solution. □

This example also demonstrates the advantages of using a PB solver. Firstly, the produced cores are not clauses but more general PB constraints, thanks to which we can obtain larger increases in the lower bound. Secondly, it is very easy to add the upper bound after a solution is found, because the upper bound is represented as a single constraint that the solver can handle natively. Finally, the PB solver can employ strong reasoning based on the cutting planes proof system.

To fully utilize the potential of combining PB optimization with core-guided search we need to overcome multiple challenges: How do we extract cores? How can we guarantee a sound reformulation for arbitrary PB cores? How do we avoid introducing huge numbers of variables during reformulation? How do we choose the set of assumptions? We will discuss these questions in what follows.

Algorithm 2 Computation of cardinality constraint implied by C with maximal lower bound increase.

```
1: procedure MAXLOWERBOUNDINCREASE(C, O)
2:     lb⁺ ← 0
3:     repeat
4:         Σ_{ℓ∈X} ℓ ≥ B ← round2card(C)
5:         m ← min_{ℓ∈X} a_ℓ
6:         if B · m > lb⁺ then
7:             lb⁺ ← B × m
8:             R ← Σ_{ℓ∈X} ℓ ≥ B
9:         C ← weaken(C, {ℓ | (m, ℓ) ∈ O})
10:    until B ≤ 0
11:    return R
```

Contributions

Lifting Incremental Solving to PB Incremental PB solving is similar to incremental SAT solving, but there are more degrees of freedom to core extraction, which we explore in this section. Similar to SAT, we detect unsatisfiability with respect to the assumptions when a learned constraint L is generated that causes a conflict after propagating the assumptions. All literals in L which were falsified by propagation can be systematically eliminated from L to generate a PB *core constraint* C.

One difference from MaxSAT is that a PB core constraint can still contain non-assumption literals. These can be safely eliminated by weakening. The resulting *decision literal core* is a PB constraint only involving assumption literals. Another difference is that it could be that a valid core constraint is encountered before all falsified non-assumption literals have been eliminated. If so, we can stop core extraction immediately and weaken all non-assumption literals to obtain a constraint that we will call an *early core*. Both of these scenarios arise fairly frequently.

Postprocessing the Core to a Cardinality Constraint The cores obtained from the solver will in general be arbitrary PB constraints. It turns out to simplify matters (as we will explain later) to round this constraint to a *cardinality core constraint*. In general, a PB constraint $C \equiv \sum a_i \ell_i \geq B$ can imply multiple cardinality constraints, so we need to choose how to round. This is another example of a question that does not arise in MaxSAT solving. We consider two options: (a) the cardinality constraint that maximizes the lower bound increase; and (b) the shortest implied clause.

Maximal lower bound increase Given a cardinality core $\sum_{\ell \in X} \ell \geq B$, let $a_\ell > 0$ be the coefficient of the literal ℓ in the objective function to be minimized. The increase in the lower bound caused by this core is $B \cdot \min_{\ell \in X} a_\ell$. To find the best such core, we weaken all literals with small coefficients and evaluate the resulting rounded cardinality constraint in terms of lower bound increase as described in Algorithm 2.

Example 2. Consider the core $C \equiv 3x_1 + 3x_2 + 2x_3 + x_4 + x_5 \geq 7$ with objective $7x_1 + 3x_2 + 9x_3 + 6x_4 + 7x_5$. Then $\mathsf{round2card}(C) = x_1 + x_2 + x_3 + x_4 + x_5 \geq 3$. We compute $m = 3$ and set $lb^+ = 9$. We then weaken C to obtain $3x_1 + 2x_3 + x_4 + x_5 \geq 4$, which can be rounded to $R \equiv x_1 + x_3 + x_4 + x_5 \geq 2$. We compute $m = 6$ and set $lb^+ = 12$ storing R as the current best. We then weaken C obtaining $3x_1 + 2x_3 + x_5 \geq 1$, which yields the cardinality constraint $x_1 + x_3 + x_5 \geq 1$. We compute $m = 7$ but lb^+ does not increase. We weaken C to obtain $2x_3 \geq -3$, at which point the loop is exited and R is returned. □

Minimal Size Clause We construct a minimal size clause from C by weakening literals with the smallest coefficients until the degree is no greater than any remaining coefficient, after which we can construct a clause by division with the greatest coefficient a_{max}.

Example 3. As in Example 2, consider the core $C \equiv 3x_1 + 3x_2 + 2x_3 + x_4 + x_5 \geq 7$. Then we can weaken x_5, x_4 and x_3 to obtain $3x_1 + 3x_2 \geq 3$, with the remaining coefficients at least the weakened degree. The resulting shortest clause is $\mathsf{divide}(3x_1 + 3x_2 \geq 3, 3)$ or $x_1 + x_2 \geq 1$. □

Objective Reformulation After we obtained a cardinality constraint $\sum_{\ell \in X} \ell \geq B$ from a core, we reformulate the objective function such that the new lower bound is reflected in the constant part of the objective. Broadly speaking, we follow the OLL approach (Andres et al. 2012; Morgado, Dodaro, and Marques-Silva 2014) in MaxSAT solving, but with the crucial difference that there is no re-encoding of cardinality constraints to clauses. Instead, we introduce fresh variables z_i and add to the solver constraint database *sum encoding* constraints

$$\sum_{\ell \in X} \ell = B + \sum_{i=B+1}^{|X|} z_i \tag{3}$$

representing the sum of the literals in the core in unary, as well as *ordering constraints* $z_i \geq z_{i+1}$ to enforce that z_i is true if and only if $\sum_{\ell \in X} \ell \geq i$ holds. Here it is important to observe that if we had a general PB constraint in (3), we could be forced to introduce an exponential number of variables. Using (3), we can reformulate the objective function as

$$\sum_{(a,\ell) \in O} a\ell + c_o \tag{4a}$$

$$= \sum_{(a,\ell) \in O} a\ell + c_o + m(\sum_{\ell \in X} \ell) - m(\sum_{\ell \in X} \ell) \tag{4b}$$

$$\overset{(3)}{=} \sum_{(a,\ell) \in O} a\ell + c_o + m(B + \sum_{i=B+1}^{|X|} z_i) - m(\sum_{\ell \in X} \ell)\,, \tag{4c}$$

where m is the smallest coefficient in the objective function of literals in X. In this way, we ensure that $\sum_{(a,\ell) \in O} a\ell - m(\sum_{\ell \in X} \ell)$ still only has non-negative coefficients. This in turn means that the constant term $c_o + mB$ in the reformulated objective function is the new lower bound. Note that it is always possible to rewrite the objective function in this way, because we only assume to false literals that appear in the current reformulated objective function. Furthermore, every reformulation strictly increases the lower bound, so we will will eventually reach an optimal lower bound.

Lazy Variable Encodings One problematic issue with the objective function reformulation as described above is that every new reformulation can introduce many fresh variables, and as the number of new variables increases this can slow down the solver in future incremental calls. To alleviate this, one can introduce the z_i variables *lazily*, i.e., only when they are needed. Observe that in view of the ordering constraints $z_i \geq z_{i+1}$ we know that setting z_k to false forces z_i to false for all $i > k$ as well. Hence, we do not need the variables z_i for $i > k$ when assuming z_k as false. (This observation was also made in the context of incremental re-encoding of cardinality constraints to clauses in (Martins et al. 2014).)

To obtain a lazy encoding we take the sum encoding (3) and remove variables in a safe manner. Assume we only want to introduce variables up to z_k. Then we can write the equality in (3) as two inequalities

$$\sum_{i=B+1}^{|X|} z_i \leq \sum_{\ell \in X} \ell - B \leq \sum_{i=B+1}^{|X|} z_i \,. \tag{5}$$

For the lower bound on the left we can just omit the variables $z_i, i > k$, because this will only make the lower bound smaller. For the upper bound on the right we compensate for the removed variables by increasing the coefficient of z_k. This leads to the *lazy sum encoding*

$$\sum_{i=B+1}^{k} z_i \leq \sum_{\ell \in X} \ell - B \leq \\ \leq \sum_{i=B+1}^{k-1} z_i + (|X| - k + 1) z_k \,. \tag{6}$$

If we also want to remove z_i for $i < k$, we can do so in the upper bound on the right by replacing them with 1, and in the lower bound on the left we increase the coefficient of z_k so that the correct bound is implied for $z_k = 1$. This results in the *lazy reified encoding*

$$(k - B) z_k \leq \sum_{\ell \in X} \ell - B \leq \\ \leq (k - B - 1) + (|X| - k + 1) z_k \,. \tag{7}$$

Note that if $z_k = 0$, this simplifies to $B \leq \sum_{\ell \in X} \ell \leq k - 1$, and if $z_k = 1$ to $k \leq \sum_{\ell \in X} \ell \leq |X|$ as desired.

When a core is found, we still use the sum encoding (3) to reformulate the objective. This is implemented by maintaining an implicit representation of the reformulated objective function, storing for each core the factor used for reformulation as well as the number of variables that have not yet been introduced due to laziness. Instead of adding constraints as in (3) to the solver database, we only add the lazy encoding for a single variable. Due to other cores this variable can disappear from the reformulated objective and at this point we add the next variable and the corresponding lazy encoding to the solver. In case of the lazy sum encoding the solver can delete constraints that were introduced for variables with smaller index.

Utilizing the Upper Bound A further improvement can be achieved if an upper bound $u \in \mathbb{N}$ is known for the literals in the core, i.e., $\sum_{\ell \in X} \ell \leq u$. Such an upper bound means that we do not need to introduce all z_i variables but only variables up to $i = \min(u, |X|)$.

Hybrid Search The simplest strategy for choosing the assumptions is to not set any assumptions at all, which results in pure *linear search*. If the set of assumptions is non-empty, we refer to this as *core-guided search*. Since adding upper and lower bound constraints can be achieved with very low overhead in a native PB setting, we explore a *hybrid search* variant where we switch back and forth between running the solver with and without assumptions, trying to roughly balance the time spent on linear and core-guided search, respectively (measuring not running time, however, but different statistics such as number of literals investigated during unit propagation, in order make the solver deterministic for reproducibility purposes).

Experiments

We have implemented the discussed core-guided techniques in the PB solver ROUNDINGSAT (Elffers and Nordström 2018) and we have evaluated our implementation on four benchmark sets (converted to the standard OPB format used for PB solvers as needed):

- PB16: OPT-SMALL-INT benchmarks from the most recent PB Competition in 2016.
- MIP: 0-1 integer linear programming optimization problems from MIPLIB.
- KNAP: Knapsack benchmarks from (Pisinger 2005).
- CRAFT: Some crafted combinatorial benchmarks.

For comparing against other PB solvers, the PB16 benchmarks are the main target. We also study MIP and KNAP because they are two quite challenging sets of benchmarks for PB solvers, as observed in (Devriendt, Gleixner, and Nordström 2021). Finally, the crafted benchmarks are inspired by (Elffers et al. 2018; Vinyals et al. 2018), but have been generated with larger parameters so as to be more challenging. This allows us to "stress-test" the solvers by exposing them to problems that provably require sophisticated reasoning.

As hardware we used AMD Opteron 6238 nodes having 6 cores and 16 GiB of memory running Ubuntu 16.04.7. Each run was executed as a single thread on a node (leaving 5 cores unused to avoid timing issues due to competition for memory resources) with a 5000 second time-out limit. Binary, source code and detailed experimental results are available online (Devriendt et al. 2020).

Contribution of Core-Guided PB Techniques In order to investigate the impact of the different core-guided techniques we have developed for PB solving, we started by running extensive experiments with a large number of different settings to identify a good base configuration. Guided by these experiments, we chose a configuration that will be referred to as HYBRID in what follows. It uses hybrid solving interleaving core-guided and linear search phases, chooses the cardinality core that yields the largest increase of the lower bound for the objective function, and reformulates the objective using the lazy reified encoding. We then investigated three technical novelties of PB core-guided search:

1. non-clausal cores (comparing with HYBRIDCLAUSAL deriving clausal cores instead of cardinality cores),

	PB16 (1600)	MIP (291)	KNAP (783)	CRAFT (985)
HYBRID	968	78	306	639
HYBRIDCLAUSAL	937	75	298	618
HYBRIDNONLAZY	936	70	186	607
HYBRIDCLNONL	917	67	203	612
ROUNDINGSAT	853	75	341	309
COREGUIDED	911	61	43	595
COREBOOSTED	959	80	344	580
SAT4J	773	61	373	105
NAPS	896	65	111	345
SCIP	1057	125	765	642

Table 1: Number of instances solved to optimality for state-of-the-art solvers and ROUNDINGSAT core-guided variants.

2. lazy reformulation of the objective function (comparing with non-lazy reformulation in HYBRIDNONLAZY),
3. hybrid optimization with repeated switches back and forth between core-guided search and linear search (compared to linear search as in standard ROUNDINGSAT, pure COREGUIDED, and COREBOOSTED approaches).

We want to stress that the language of PB inequalities gives native support for an efficient implementation of this kind of approaches, in contrast to CNF. The top seven configurations of Table 1 shows that all three new features listed above significantly improve PB solver performance.

In more detail, Figure 1 provides a scatter plot of the number of cores needed to prove optimality for the configuration HYBRID as compared to the version HYBRIDCLAUSAL with clausal cores, *except that to get as clear a comparison as possible the plots are for pure core-guided search* with these solvers—adding linear search does not change the conclusions, but just makes the plot more fuzzy. Clearly, in the non-clausal settings fewer cores are needed.

In Figure 2 we study the number of new variables introduced by HYBRID as compared to the version HYBRIDNONLAZY that eagerly introduces all new variables in one go when the objective is reformulated. For many instances the lazy approach introduces orders of magnitude less variables, and this effect is especially pronounced for knapsack instances. We studied the two different lazy encodings (lazy sum and lazy reified) but did not see any significant differences in performance between the two—what is important is to avoid non-lazy reformulation. It is worth noting that even the weakest core-guided configuration HYBRIDCLNONL with clausal cores and non-lazy reformulations is clearly better than the original ROUNDINGSAT solver on the PB16 benchmarks, so core-guided search in itself is powerful.

The theoretical benchmarks in CRAFT give us a possibility to peek inside the solver, as it were, by exposing it to formulas expressing different combinatorial principles and thus requiring different forms of sophisticated reasoning. It is striking that on these benchmarks we see the clearest gains from the core-guided techniques.

Overall, a clear message is that adding core-guided techniques provides a dramatic boost for PB solving. And even

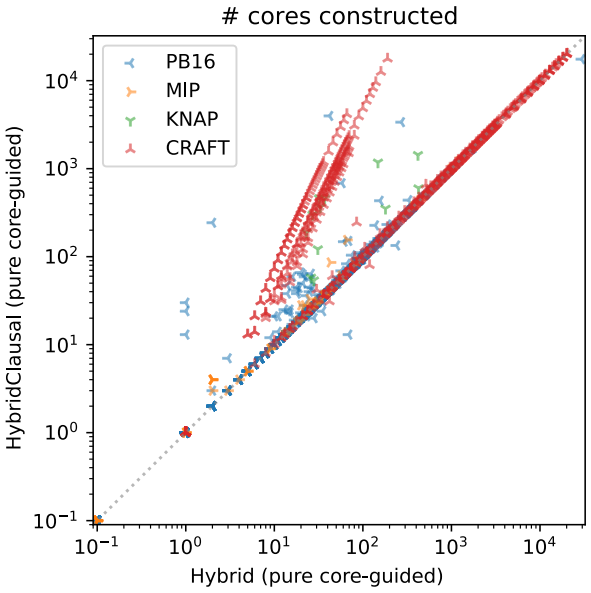


Figure 1: Number of cores during search for non-clausal (x-axis) versus clausal (y-axis) cores for instances solved by both approaches (but using pure core-guided optimization).

though the simplest version of core-guided search, without exploiting PB-specific techniques, can already provide major gains for some domains compared to the non-core-guided solver, our further PB optimizations help significantly to give more consistent performance improvements.

An interesting question is how to balance lower-bounding search using core-guided solving and upper-bounding linear search. As can be seen in Table 1, pure core-guided search (COREGUIDED) is not universally beneficial, and for KNAP even the hybrid mode is clearly not helpful compared to simple linear search. But it is interesting that our configuration COREBOOSTED with 10% *core-boosting* (Berg, Demirović, and Stuckey 2019) shows that a little bit of core-guided search can also help on these benchmarks. Overall, our new hybrid mode, switching repeatedly between core-guided and linear search, is the best. Importantly, this is not just an effect of hybrid providing a portfolio, as it were, of pure core-guided and pure linear search. For the crafted benchmarks, we verified that the hybrid solver even beats a parallel version where pure core-guided and pure linear search get to run side by side, each with a 5000 second time-out.

We have also evaluated the popular heuristics *stratification* (Ansótegui et al. 2012) and *independent cores* (Berg and Järvisalo 2017) from the core-guided MaxSAT literature. Figure 3 shows the effects of turning on stratification (HYBRIDSTRAT) and independent cores (HYBRIDSTRATIND) for the PB16 benchmarks. Looking at all benchmarks, switching on both stratification and independent cores helps for MIP and KNAP but does not change much for PB16 and is terrible for CRAFT. Stratification alone seems never to be a bad idea—we could have included it in our base configuration HYBRID and the con-

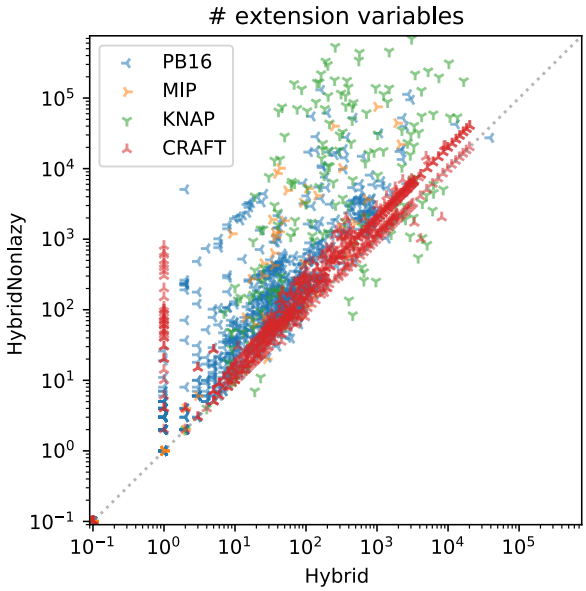


Figure 2: Number of new variables during search for lazy (x-axis) versus non-lazy (y-axis) objective reformulation for instances solved by both approaches.

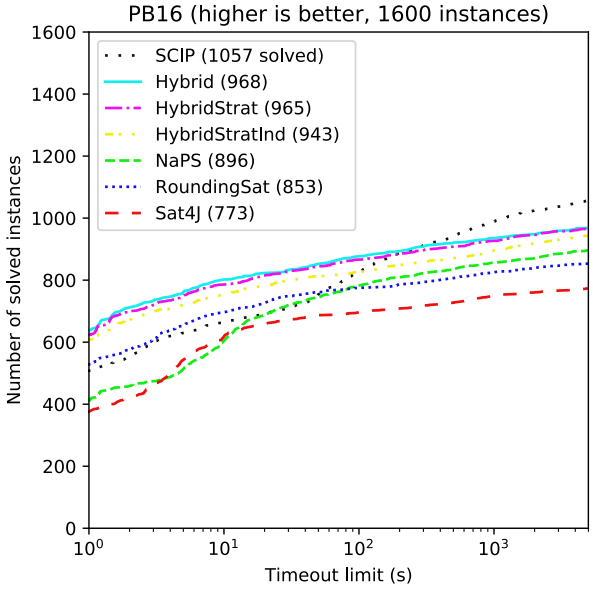


Figure 3: Cumulative plot for PB16 benchmarks.

clusions would not really have changed—but using independent cores can cause real problems for the wrong kind of benchmarks, which is especially clear for CRAFT. We are currently unable to explain why this is so.

Comparison to State of the Art In addition to comparing our core-guided PB solver to the original RoundingSat version, we evaluate two other PB solvers that performed well in the PB16 Competition as well as one MIP solver:

- Sat4J (Le Berre and Parrain 2010) commit c091d768. We use the *Both* strategy that essentially runs a CDCL solver and a cutting-planes-based PB solver in parallel.[1]
- NaPS (Sakai and Nabeshima 2015) commit 7aaa54f4. We use the *bignum* version as suggested to us by the authors. In contrast to RoundingSat and Sat4J, NaPS does *not* use cutting-planes-based reasoning but instead re-encodes the input to CNF and runs a CDCL solver.
- SCIP (Gamrath et al. 2020) version 7.0.0 using SoPlex version 5.0.0 as LP solver, with presolving support of Papilo 1.0 but without symmetry detection.

We present the results of this comparison in Table 1. Figure 3 gives a more detailed picture of the PB16 benchmarks.

Overall, our core-guided PB solver Hybrid decisively beats RoundingSat, Sat4J, and NaPS, with the notable exception that the dual-threaded version of Sat4J is best for KNAP benchmarks.

Sadly, PB solvers still struggle to compete with MIP solvers such as SCIP, and addressing this shortcoming seems to be the most interesting challenge for future research. One important factor to note is that presolving is a very important part of MIP performance, whereas current cutting-planes-based PB solvers essentially have no preprocessing. A natural approach would be to integrate the Papilo presolver with a cutting-planes-based PB solver and see what happens to performance. Another direction would be to combine core-guided solving with the use of linear programming relaxations, which is another core component of MIP solvers, and where the results in (Devriendt, Gleixner, and Nordström 2021) look promising. Already now, though, the results for some of the benchmarks in CRAFT show that there are problems that PB solvers solve very efficiently but that are beyond MIP solvers such as SCIP.

[1]This dual-threaded approach gets twice the CPU time of the other solvers, but we kept it so that our core-guided PB solvers would compete against the best version of Sat4J for each instance.

Concluding Remarks

In this work, we extend the resolution-based core-guided approach to pseudo-Boolean solvers using cutting planes reasoning, and show that this leads to dramatic improvements. The fact that PB solvers have native handling of PB constraints make them a very good fit for core-guided search.

Our work opens several directions for future work. One important problem is to find better strategies for balancing the lower- and upper-bounding phases in our hybrid approach for PB solvers. Independent core extraction is another technique that seems worth studying more closely—it has been successful for MaxSAT and CP solvers, but we see no such clear gains, and on the contrary this approach is largely detrimental for crafted instances. Finally, we would like to understand why MIP solvers are often (though not always!) much better than the best core-guided PB approach. Since preprocessing and LP relaxations are important components in MIP solvers, employing these techniques in PB core-guided search may be one key to further gains.

Acknowledgments

The first, second and fourth author were supported by the Swedish Research Council grant 2016-00782, and the fourth author also received funding from the Independent Research Fund Denmark grant 9040-00389B. Furthermore, the work was partially supported by the University of Melbourne AI Research group funding.

References

Achterberg, T. 2007. Conflict Analysis in Mixed Integer Programming. *Discrete Optimization* 4(1): 4–20.

Alviano, M.; Dodaro, C.; Marques-Silva, J. P.; and Ricca, F. 2015. Optimum Stable Model Search: Algorithms and Implementation. *Journal of Logic and Computation* 30(4): 863–897.

Alviano, M.; Dodaro, C.; and Ricca, F. 2015. A MaxSAT Algorithm Using Cardinality Constraints of Bounded Size. In *Proceedings of the 24th International Conference on Artificial Intelligence (IJCAI '15)*, 2677—2683.

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. 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)*, 211–221.

Ansótegui, C.; Bonet, M. L.; Gabàs, J.; and Levy, J. 2012. 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*, 86–101. Springer.

Bayardo Jr., R. J.; and Schrag, R. 1997. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, 203–208.

Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Towards Understanding and Harnessing the Potential of Clause Learning. *Journal of Artificial Intelligence Research* 22: 319–351. Preliminary version in *IJCAI '03*.

Berg, J.; Demirović, E.; and Stuckey, P. J. 2019. Core-Boosted Linear Search for Incomplete MaxSAT. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR '19)*, volume 11494 of *Lecture Notes in Computer Science*, 39–56. Springer.

Berg, J.; and Järvisalo, M. 2017. 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*, 652–670. Springer.

Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds. 2021. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2nd edition.

Buss, S. R.; and Nordström, J. 2021. Proof Complexity and SAT Solving. In (Biere et al. 2021), chapter 7, 233–350.

Chai, D.; and Kuehlmann, A. 2005. A Fast Pseudo-Boolean Constraint Solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(3): 305–317. Preliminary version in *DAC '03*.

Cook, S. A. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, 151–158.

Cook, W.; Coullard, C. R.; and Turán, G. 1987. On the Complexity of Cutting-Plane Proofs. *Discrete Applied Mathematics* 18(1): 25–38.

Demirović, E.; Chu, G.; and Stuckey, P. J. 2018. Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers. In *International Conference on Principles and Practice of Constraint Programming*, 99–108. Springer.

Demirović, E.; and Stuckey, P. J. 2019. Techniques Inspired by Local Search for Incomplete MaxSAT and the Linear Algorithm: Varying Resolution and Solution-Guided Search. In *International Conference on Principles and Practice of Constraint Programming*, 177–194. Springer.

Devriendt, J.; Gleixner, A.; and Nordström, J. 2021. Learn to Relax: Integrating 0-1 Integer Linear Programming with Pseudo-Boolean Conflict-Driven Search. *Constraints* Preliminary version in *CPAIOR '20*.

Devriendt, J.; Gocht, S.; Demirović, E.; Nordström, J.; and Stuckey, P. J. 2020. Experimental Repository for "Cutting to the Core of Pseudo-Boolean Optimization: Combining Core- Guided Search with Cutting Planes Reasoning". URL https://doi.org/10.5281/zenodo.4043124.

Eén, N.; and Sörensson, N. 2003. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science* 89(4): 543–560.

Elffers, J.; Giráldez-Cru, J.; Nordström, J.; and Vinyals, M. 2018. 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*, 75–93. Springer.

Elffers, J.; and Nordström, J. 2018. Divide and Conquer: Towards Faster Pseudo-Boolean Solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI '18)*, 1291–1299.

Fu, Z.; and Malik, S. 2006. 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*, 252–265. Springer.

Gamrath, G.; Anderson, D.; Bestuzheva, K.; Chen, W.-K.; Eifler, L.; Gasse, M.; Gemander, P.; Gleixner, A.; Gottwald, L.; Halbig, K.; Hendel, G.; Hojny, C.; Koch, T.; Bodic, P. L.; Maher, S. J.; Matter, F.; Miltenberger, M.; Mühmer, E.; Müller, B.; Pfetsch, M.; Schlösser, F.; Serrano, F.; Shinano, Y.; Tawfik, C.; Vigerske, S.; Wegscheider, F.; Weninger, D.; and Witzig, J. 2020. The SCIP Optimization Suite 7.0. Technical report, Optimization Online. URL http://www.optimization-online.org/DB_HTML/2020/03/7705.html.

Gange, G.; Berg, J.; Demirović, E.; and Stuckey, P. J. 2020. 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*, 205–221. Springer.

Haken, A. 1985. The Intractability of Resolution. *Theoretical Computer Science* 39(2-3): 297–308.

Le Berre, D.; and Parrain, A. 2010. The Sat4j Library, Release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7: 59–64.

Levin, L. A. 1973. Universal Sequential Search Problems. *Problemy peredachi informatsii* 9(3): 115–116. In Russian. Available at http://mi.mathnet.ru/ppi914.

Manquinho, V. M.; Marques-Silva, J. P.; and Planes, J. 2009. Algorithms for Weighted Boolean Optimization. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT '09)*, volume 5584 of *Lecture Notes in Computer Science*, 495–508. Springer.

Marques-Silva, J. P.; and Sakallah, K. A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers* 48(5): 506–521. Preliminary version in *ICCAD '96*.

Martins, R.; Joshi, S.; Manquinho, V. M.; and Lynce, I. 2014. 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*, 531–548. Springer.

Morgado, A.; Dodaro, C.; and Marques-Silva, J. P. 2014. 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*, 564–573. Springer.

Morgado, A.; Heras, F.; Liffiton, M.; Planes, J.; and Marques-Silva, J. 2013. Iterative and Core-Guided MaxSAT Solving: A Survey and Assessment. *Constraints* 18: 478–534.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, 530–535.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via Lazy Clause Generation. *Constraints* 14(3): 357–391.

Pipatsrisawat, K.; and Darwiche, A. 2007. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT '07)*, volume 4501 of *Lecture Notes in Computer Science*, 294–299. Springer.

Pisinger, D. 2005. Where are the hard knapsack problems? *Computers & Operations Research* 32(9): 2271–2284.

Sakai, M.; and Nabeshima, H. 2015. 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.

Sheini, H. M.; and Sakallah, K. A. 2006. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation* 2(1-4): 165–189. Preliminary version in *DATE '05*.

Stuckey, P. J. 2010. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR '10)*, volume 6140 of *Lecture Notes in Computer Science*, 5–9. Springer.

Vinyals, M.; Elffers, J.; Giráldez-Cru, J.; Gocht, S.; and Nordström, J. 2018. In Between Resolution and Cutting Planes: A Study of Proof Systems for Pseudo-Boolean SAT Solving. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT '18)*, volume 10929 of *Lecture Notes in Computer Science*, 292–310. Springer.

Paper D

Certified CNF Translations for Pseudo-Boolean Solving

Stephan Gocht
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

Ruben Martins
Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

Jakob Nordström
University of Copenhagen, Copenhagen, Denmark
Lund University, Lund, Sweden

Andy Oertel
Lund University, Lund, Sweden
University of Copenhagen, Copenhagen, Denmark

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.

2012 ACM Subject Classification Theory of computation → Program verification; Hardware → Theorem proving and SAT solving; Theory of computation → Logic and verification

Keywords and phrases pseudo-Boolean solving, 0-1 integer linear program, proof logging, certifying algorithms, certified translation, CNF encoding, cutting planes

Funding *Stephan Gocht*: Swedish Research Council grant 2016-00782.
Ruben Martins: National Science Foundation award CCF-1762363 and Amazon Research Award.
Jakob Nordström: Swedish Research Council grant 2016-00782 and Independent Research Fund Denmark grant 9040-00389B.
Andy Oertel: Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

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)* [36, 39], and this has led to a wide range of applications including large-scale problems in both academia and industry [8]. The conflict-driven paradigm has also been successfully exported to other areas such as *maximum satisfiability (MaxSAT)*, *pseudo-Boolean (PB) solving*,

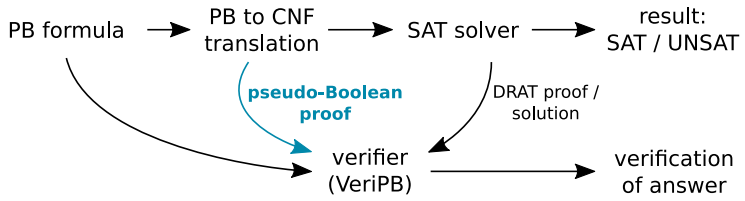


Figure 1 Proof logging workflow for pseudo-Boolean solving (our contribution in blue boldface).

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 [1, 14, 24]. For SAT solvers, however, analogous problems [9] have been successfully addressed by the introduction of *proof logging*, requiring that solvers should be *certifying* [37] 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* [28, 47], *TraceCheck* [7], *DRAT* [29, 30, 50], *GRIT* [16], and *LRAT* [15]. Since 2013 the SAT competitions [45] 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 fact that they are limited 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 in conjunctive normal form (CNF). One way to address this problem could be to allow extensions to the *DRAT* format [2], but another approach pursued in recent years is to develop stronger proof logging methods based on other formalisms such as binary decision diagrams [4], algebraic reasoning [33, 44], pseudo-Boolean reasoning [21, 25, 26], or integer linear programming [12, 19].

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+* [18]. The two solvers *Open-WBO* [41] and *NaPS* [40] using this approach were among the top performers in the latest pseudo-Boolean evaluation [43]. While *DRAT* proof logging can be used to certify unsatisfiability of the translated formula, it cannot prove correctness of the translation, not only since there is no known method of carrying out pseudo-Boolean reasoning efficiently in *DRAT* (except for constraints with small coefficients [10]), but also, and more fundamentally, because the input is not in CNF.

We demonstrate how to instead use the *cutting planes* proof method [13], enhanced with a rule allowing to introduce extension variables [27], 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) to achieve end-to-end verification of the pseudo-Boolean solving process using the proof checker *VeriPB* [48], as illustrated in Figure 1.

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., [42]), 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 encoding of the computation of this circuit. An important part of our contribution is that we develop a general framework to provide proof logging for a wide class of such circuits in a uniform way. The pseudo-Boolean format used for proof logging makes it easy to derive 0-1 linear inequalities describing the computations in the circuit, and once this has been done the desired clauses in the CNF translation can simply be obtained by so-called *reverse unit propagation (RUP)* [28, 47], obviating the need for complicated syntactic proofs. We have applied this method to the *sequential counter* [46], *totalizer* [3], *generalized totalizer* [32] and *adder network* [18, 49] encodings, and report results from an empirical evaluation.

We note that a slightly stronger result would be to certify *equivalence* of the original pseudo-Boolean formula F and the translated CNF formula F', in the sense that 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 any satisfying assignment α' to F' also satisfies F. The tools we develop can reach this more ambitious goal in principle, but some additional technical work is required, due to which we 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 briefly discuss how to apply it to adder networks in Section 5. (Details for the totalizer and generalized totalizer encodings are in the appendix.) We report experimental data for proof logging and verification in Section 6 and conclude with a discussion of some possible directions for future research in Section 7.

2 Preliminaries

Let us start with a review of some standard material that can also be found in, e.g., [11, 27]. 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 sometimes write $\vec{x} = \{ x_1, \dots, x_m \}$ to denote a set of variables. A *pseudo-Boolean (PB) constraint* is a 0-1 linear inequality

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

which without loss of generality we always assume to be in *normalized form* [5]; 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

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

An equality constraint $C \doteq \sum_i a_i \ell_i = A$ is just syntactic sugar for the pair of inequalities $C^{\text{geq}} \doteq \sum_i a_i \ell_i \geq A$ and $C^{\text{leq}} \doteq \sum_i -a_i \ell_i \geq -A$ (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 $F \doteq \bigwedge_j C_j$ of PB constraints. A *cardinality constraint* is a PB constraint with all coefficients equal to 1. If the degree of falsity is also 1, then the constraint $\ell_1 + \dots + \ell_k \geq 1$ is equivalent to the *(disjunctive) clause* $\ell_1 \vee \dots \vee \ell_k$, and so CNF formulas are just special cases of PB formulas.

A *(partial) assignment* ρ is a (partial) function from variables to $\{ 0, 1 \}$. 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 $F{\restriction_\rho} \doteq \bigwedge_j C_j{\restriction_\rho}$. 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 [13] 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 added. (When referring to a linear combination of two equality constraints C and D, we mean the linear combinations of C^{geq} and D^{geq} and C^{leq} and D^{leq}, respectively, with the same positive integer factors.) 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 $\sum_i a_i \cdot \ell_i \geq A$.

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.

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.

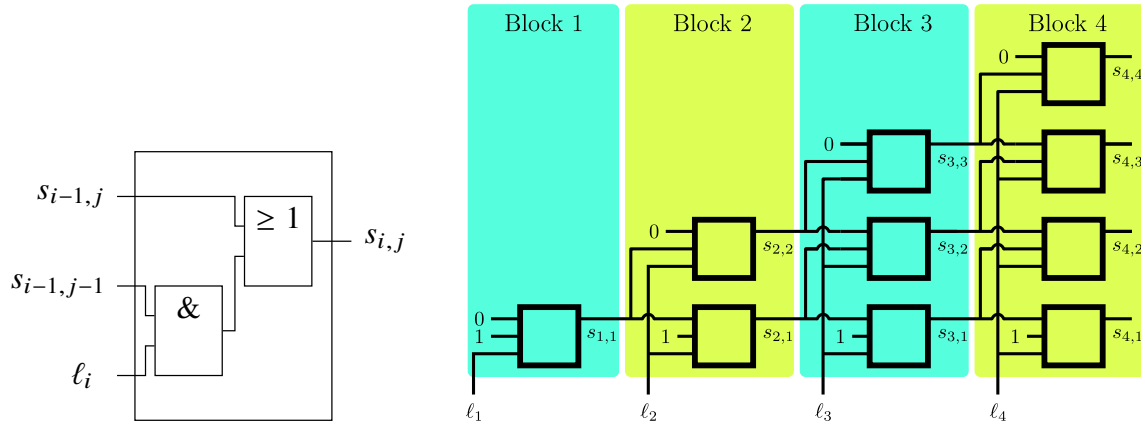
In addition to deriving constraints C that are implied by F, we will also need a rule for adding so-called *redundant* constraints D having the property that F and $F \wedge D$ are equisatisfiable. For this purpose we will use the *reification* rules described below, which are shown in [27] to be special cases of the redundance rule in that paper. Provided that z is a *fresh variable* that is not in the formula and has not appeared previously in the derivation, we can introduce the *reified constraints*

$$z \Rightarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad A\overline{z} + \sum_i a_i \ell_i \geq A \tag{3a}$$

and

$$z \Leftarrow \textstyle\sum_i a_i \ell_i \geq A \quad \doteq \quad \big(\sum_i a_i - A + 1\big) \cdot z + \sum_i a_i \overline{\ell}_i \geq \sum_i a_i - A + 1\,. \tag{3b}$$

A moment of thought reveals that the constraint (3a) 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 (3b) 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 (3a) and (3b). It is easy to see that adding such reification constraints to a formula F preserves equisatisfiability, since any satisfying assignment to F can be extended by setting z as required to satisfy the implications.



(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.

3 Certified Translation for the Sequential Counter Encoding

To encode a cardinality constraint of the form $\sum_{i=1}^{n} \ell_i \bowtie k$ we can use the *sequential counter encoding* [46]. This encoding is designed after a circuit accumulating the sum of input bits using the intermediate fresh variables $s_{i,j}$ for $i \in [n], j \in [i]$, where $s_{i,j}$ is true if and only if the first i literals sum up to j. The variable $s_{i,j}$ is computed as in Figure 2a, i.e.,

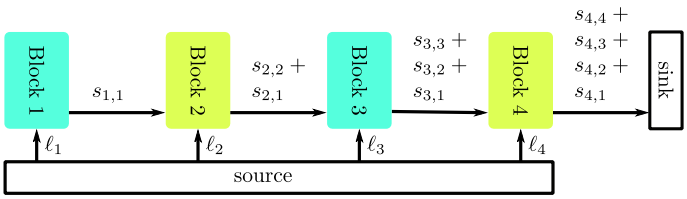
$$s_{i,j} \leftrightarrow ((\ell_i \wedge s_{i-1,j-1}) \vee s_{i-1,j}) \,, \tag{4}$$

that is either the first $i-1$ variables add up to $j-1$ and the i-th literal is true, or the first $i-1$ variables already add up to j. The resulting circuit is shown in Figure 2b and can be divided into multiple blocks, where the i-th block accumulates the i-th input literal and the variables $s_{i-1,j}$ for $j \in [i-1]$. We will use this block structure later as an abstract way to represent the encoding. The clausal encoding is given by translating the circuit into clausal form, i.e., via the clauses

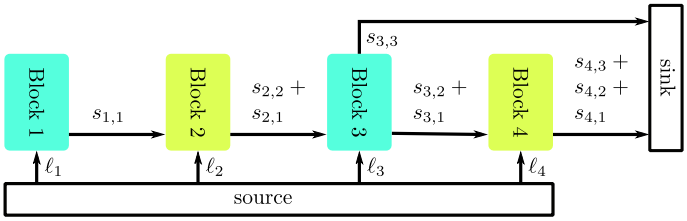
$$\bar{\ell}_i + \bar{s}_{i-1,j-1} + s_{i,j} \geq 1 \tag{5a}$$
$$\bar{s}_{i-1,j} + s_{i,j} \geq 1 \tag{5b}$$
$$\ell_i + s_{i-1,j} + \bar{s}_{i,j} \geq 1 \tag{5c}$$
$$s_{i-1,j-1} + \bar{s}_{i,j} \geq 1 \,, \tag{5d}$$

where $i \in [n]$ and $j \in [i]$. To cover corner cases we always replace $s_{i,j}$ for $j > i$ with 0 and $s_{i,j}$ for $j \leq 0$ with 1 and simplify the constraints accordingly. For example, for $i = j = 1$ we only get the clauses $\bar{\ell}_1 + s_{1,1} \geq 1$ and $\ell_1 + \bar{s}_{1,1} \geq 1$, since $s_{0,0}$ is replaced by 1 and hence the variable disappears from (5a) while (5d) is satisfied, and $s_{0,1}$ is replaced by 0 and thus disappears from (5c) and satisfies (5b). To enforce a greater-or-equal-k constraint it is only necessary to add the clause $s_{n,k} \geq 1$. Analogously, a less-or-equal-k constraint is enforced using the clause $\bar{s}_{n,k+1} \geq 1$. A common optimization, known as k-simplification, is to not add the clauses for variable $s_{i,j}$ if $j > k+1$, as these variables have no influence on the satisfiability of the clausal encoding.

Before discussing the proof logging, let us study the encoding in more detail, ignoring k-simplification for now. Remember that the variable $s_{i,j}$ should be true if and only if the first i literals sum up to at least j and hence can be understood as a unary representation, where we want that $\sum_{i=1}^{i} \ell_j = \sum_{i=1}^{i} s_{i,j}$ for $i \in [n]$. However, the circuit is only using



(a) Graph without k-simplification.



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

Figure 3 Graph representation of the sequential counter encoding.

the variables from the previous block $s_{i-1,j}$ and the literal ℓ_i as input to compute the $s_{i,j}$ variables and hence it will instead be more convenient to consider the equality

$$\ell_i + \sum_{j=1}^{i-1} s_{i-1,j} = \sum_{j=1}^{i} s_{i,j} \quad i \in [n] \ . \tag{6}$$

We can use this insight to get a more abstract representation of the circuit in Figure 2b, by thinking of blocks as nodes with two input edges labelled ℓ_i and $\sum_{j=1}^{i-1} s_{i-1,j}$ and an output edge labelled $\sum_{j=1}^{i} s_{i,j}$ as shown in Figure 3a. Additionally, for each inner node the sum of all input labels should be equal to the sum of all output labels as enforced by (6), which we will call a *preserving equality*. This graph representation will be helpful to generalize the presented proof logging approach for other encodings.

Note that the sum of input variables coming from the source equals the sum of output variables on the edges going to the sink because each node preserves equality between incoming and outgoing values. That is we have $\sum_{j=1}^{n} \ell_j = \sum_{j=1}^{n} s_{n,j}$, which can also be obtained mathematically by summing all equalities of the form (6). Based on this equality, it is clear that a bound on the input variables $k \bowtie \sum_{j=1}^{n} \ell_i$ also implies a bound on the output variables, which can be seen by summing $k \bowtie \sum_{j=1}^{n} \ell_i$ and $\sum_{j=1}^{n} \ell_i = \sum_{j=1}^{n} s_{n,j}$ to get

$$k \bowtie \sum_{j=1}^{n} s_{n,j} \ . \tag{7}$$

Another important observation is that the variables $s_{i,j}$ should not just take any value satisfying (6), but they should also be ordered, that is if $s_{i,j+1}$ is true, the sum should be at least $j + 1$ and hence also at least j and $s_{i,j}$ should be true as well (and also $s_{i,j-1} = 1, s_{i,j-2} = 1$, etc.). This can be enforced with *ordering constraints*

$$s_{i,j} \geq s_{i,j+1} \quad i \in [n] \, , j \in [i-1] \ . \tag{8}$$

With this improved understanding of the encoding, we can now tackle the task of proof logging, which becomes surprisingly simple. The constraints (6), (7), (8) are all pseudo-Boolean constraints and if we are able to derive them, then the clauses of the sequential counter encoding ((5) and $\bar{s}_{n,k+1} \geq 1$ and/or $s_{n,k} \geq 1$) can all be derived via reverse unit propagation: The propagations due to (8) will cause enough variables to propagate, such that (6) is falsified. The derivation of (7) from (6) was already discussed when introducing (7), where we summed all constraints (6) and the constraint to be encoded. This summation can

be expressed directly in cutting planes. For deriving the other constraints, remember that for proof logging we want to demonstrate that adding constraints does not change satisfiability. However, it is easy to see that the preserving equality (6) and ordering constraints (8) can always be satisfied by choosing a suitable value for the $s_{i,j}$ variables. If the constraints are added in ascending order of i, then the $s_{i,j}$ are fresh and can indeed be chosen freely. In the proof format this reasoning is expressed through reification as discussed in the next example and for the general case in Appendix A.1.

▶ **Example 1.** Let us consider how to derive the preserving equality

$$\ell_3 + s_{2,1} + s_{2,2} = s_{3,1} + s_{3,2} + s_{3,3} \tag{9}$$

for Block 3 in Figure 3a. To satisfy (9) we want that $s_{3,1}$ is true if $\ell_3 + s_{2,1} + s_{2,2}$ is greater equal 1, $s_{3,2}$ is true if it is greater equal 2 and $s_{3,3}$ is true if it is greater equal 3. We can enforce these conditions by introducing the fresh variables $s_{3,1}, s_{3,2}, s_{3,3}$ via reification, i.e., $s_{3,1} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 1$, $s_{3,2} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 2$ and $s_{3,3} \Leftrightarrow \ell_3 + s_{2,1} + s_{2,2} \geq 3$, which results in the pseudo-Boolean constraints

$$\overline{s}_{3,1} + \ell_3 + s_{2,1} + s_{2,2} \geq 1 \tag{10a}$$
$$2\overline{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2 \tag{10b}$$
$$3\overline{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3 \tag{10c}$$
$$3s_{3,1} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 3 \tag{10d}$$
$$2s_{3,2} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 2 \tag{10e}$$
$$s_{3,3} + \overline{\ell}_3 + \overline{s}_{2,1} + \overline{s}_{2,2} \geq 1\,. \tag{10f}$$

By design, (10a)-(10f) implies (9) and hence (9) can be derived via cutting planes. To do so in practice, we accumulate the constraints (10a)-(10c) while maintaining the invariant $\sum_{j=1}^{i} \overline{s}_{3,j} + \ell_3 + s_{2,1} + s_{2,2} \geq i$, where $i = 1, 2, 3$ is the number of accumulated constraints. When starting with (10a) the invariant holds. Next we add (10b) and divide by 2 to obtain $\overline{s}_{3,1} + \overline{s}_{3,2} + \ell_3 + s_{2,1} + s_{2,2} \geq 2$ and continue by multiplying with 2, adding (10c) and dividing by 3, which results in $\overline{s}_{3,1} + \overline{s}_{3,2} + \overline{s}_{3,3} + \ell_3 + s_{2,1} + s_{2,2} \geq 3$, which is equivalent to $\ell_3 + s_{2,1} + s_{2,2} \geq \overline{s}_{3,1} + \overline{s}_{3,2} + \overline{s}_{3,3}$, as desired. Analogously, we can accumulate (10d)-(10f) in reverse order to obtain $\ell_3 + s_{2,1} + s_{2,2} \leq \overline{s}_{3,1} + \overline{s}_{3,2} + \overline{s}_{3,3}$. The ordering constraints $s_{3,1} \geq s_{3,2}$ can be obtained by adding (10d) and (10b), which yields $3s_{3,1} + 2\overline{s}_{3,2} \geq 1$ and can be divided by 3 to obtain $s_{3,1} + \overline{s}_{3,2} \geq 1$, which is equivalent to $s_{3,1} \geq s_{3,2}$, as desired. Analogously, we can obtain $s_{3,2} \geq s_{3,3}$ by using (10e) and (10c).

To perform k-simplification, we could simply omit deriving the unneeded clauses, however this potentially introduces a large overhead for proof logging if k is small, as we would always introduce $O(n^2)$ intermediate variables instead of the $O(kn)$ variables that are needed. To avoid this overhead, as demonstrated in Figure 3b, we want that the edge going to the next block is labelled with $\sum_{j=1}^{k+1} s_{i,j}$ instead of $\sum_{j=1}^{i} s_{i,j}$. However, this means we need to introduce an additional edge going directly to the sink with the label $s_{i,k+2}$ to preserve the equality of in- and output, i.e.,

$$\ell_i + \textstyle\sum_{j=1}^{k+1} s_{i-1,j} = \sum_{j=1}^{k+2} s_{i,j} \quad i \in [n]\,. \tag{11}$$

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 $k+1$ fresh variable on the right hand side.

▶ **Example 2.** To demonstrate k-simplification, consider Block 3 in Figure 3b, which has input edges with labels $s_{2,1} + s_{2,2}$ and ℓ_3 and let us perform 1-simplification. The output of Block 3 to Block 4 should only contain the 2 variables $s_{3,1} + s_{3,2}$. To preserve equality of in- and output, we add an edge from Block 3 to the sink labelled $s_{3,3}$.

As before, we can obtain that in- and output of the graph are equal by summing the preserving constraint (11) of each node, which yields $\sum_{i=1}^{n} \left(\ell_i + \sum_{j=1}^{k+1} s_{i-1,j}\right) = \sum_{i=1}^{n} \left(\sum_{j=1}^{k+2} s_{i,j}\right)$ and can be simplified to $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} s_{i,k+2} + \sum_{j=1}^{k+1} s_{n,j}$.

4 General Framework for Certifying CNF Translations

A major challenge of providing proof logging for translations of pseudo-Boolean constraints to CNF is that there are so many different encodings of pseudo-Boolean constraints. To support a wide range of encodings, we can generalize the idea of the graph representation used in the previous section to obtain a general framework. The main ingredient of the framework is a graph representing the connection between the variables of the encoded constraint and auxiliary variables used in the encoding. This graph has the property that we can derive a preserving equality of in- and output for each node and that the CNF encoding follows from these equalities. To derive the preserving equality, we provide proof logging for general purpose operations for different ways to represent natural numbers. Let us start with a formal definition of the graph representation.

▶ **Definition 3** (Arithmetic Graph). *An* arithmetic graph *with input* $\sum_i a_i x_i$ *and output* $\sum_i c_i o_i$ *is a directed graph* $G = (V, E)$ *with a source node* s*, a sink node* t*, and edge labels of the form* $\sum_i b_i^e y_i^e$ *for each edge* $e \in E$*. For convenience, we allow to have multiple edges between two nodes. Additionally, we require that*

- *the source* s *has only outgoing edges and the input is split among edges of* s*, i.e.,* $\sum_i a_i x_i \equiv \sum_{(s,v)=e\in E} \sum_i b_i^e y_i^e$*,*
- *the sink* t *has only incoming edges and the output is split among edges of* t*, i.e.,* $\sum_i c_i o_i \equiv \sum_{(v,t)=e\in E} \sum_i b_i^e y_i^e$*, and*
- *for every inner node* v *the input is equal to the output, which can be derived via proof logging, i.e., we can derive the preserving equality*

$$\sum_{(u,v)=e\in E} \sum_i b_i^e y_i^e = \sum_{(v,u)=e\in E} \sum_i b_i^e y_i^e \,. \tag{12}$$

The general strategy for providing proof logging will be to formulate the used encoding in terms of an arithmetic graph, where the preserving equality (12) will depend on the representation of natural numbers used in the encoding and will be derived using one of the operations described later in this section. For each encoding, we will make sure that the clauses in the encoding directly correspond to a node in the graph and will follow by reverse unit propagation from the preserving equality (12). However, each encoding has also clauses to restrict the output variables o_i, which can only be derived after translating the bound known on the input variables to a bound on the output variables.

▶ **Proposition 4.** *Given an arithmetic graph with input* $\sum_i a_i x_i$ *and output* $\sum_i c_i o_i$ *and a pseudo-Boolean constraint* $\sum_i a_i x_i \bowtie k$*, where* $\bowtie \in \{\geq, \leq, =\}$*, we can derive* $\sum_i c_i o_i \bowtie k$ *using cutting planes.*

Proof. As we have an arithmetic graph, we know that we can derive (12) for every inner node in the graph. By adding all these constraints together, we obtain the constraint $\sum_i a_i x_i = \sum_i c_i o_i$, which can be combined with $\sum_i a_i x_i \bowtie k$ to obtain $\sum_i c_i o_i \bowtie k$. ◀

Algorithm 1 General algorithm for proof logging arithmetic encodings.

procedure proof_log_encoding(C, f, G, F)
  ▷ input: C is of the form $\sum_{i=1}^{n} a_i \ell_i \bowtie k$, with $k, n \in \mathbb{N}$ and $\bowtie \in \{\geq, \leq, =\}$.
  ▷ input: an arithmetic graph $G = (V, E)$ with input $\sum_i a_i x_i$ and output $\sum_i c_i o_i$
  ▷ input: a function f that takes a node and derives its preserving equality
  ▷ input: the CNF encoding F to be derived
  sum the constraints $f(v)$ for $v \in V$ in topological order to obtain $\sum_i a_i x_i = \sum_i c_i o_i$
  combine $\sum_i a_i x_i = \sum_i c_i o_i$ and C to obtain $\sum_i c_i o_i \bowtie k$
  derive each clause in the CNF encoding F via RUP

Once the bound on the input variables 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 algorithm for proof logging encodings shown in Algorithm 1. Note that the nodes of the graph need to be traversed in a topological order when deriving the preserving equality. Otherwise we can not use that the output variables of a node are fresh, which will be crucial for the presented derivations.

Let us now discuss three common ways to represent natural numbers, as well as some general purpose operations on these representations that are used to derive the preserving equality for inner nodes. The easiest way to encode a natural number j with domain $A = \{0, 1, \ldots, m\}$ using Boolean variables is to use a unary number, where the number of variables z_i set to true is equal to j, i.e., $j = \sum_{i \in [m]} z_i$. For better propagation behaviour, it is usually required that the z_i variables are ordered via constraints $z_i \geq z_{i+1}$, which enforces that z_i is true if and only if $j \geq i$. This representation is used in the sequential counter [46] and totalizer encoding [3] and is known as *order encoding*.

▶ **Proposition 5** (Unary Sum). *For any literals $\ell_1, \ldots, \ell_n$ we can derive the constraints*

$$\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i \tag{13}$$

$$z_i \geq z_{i+1} \quad i \in [n-1] \,. \tag{14}$$

using $O(n)$ steps, where $z_1, \ldots, z_n$ are fresh variables.

Conceptually, adding these constraints does not change satisfiability, because they can always be satisfied using the fresh variables. We already discussed deriving these constraints in the context of the sequential counter encoding. The general idea is to introduce the fresh variables via reification $z_i \Leftrightarrow \sum_{i=1}^{n} \ell_i \geq i$, after which we can obtain the greater-than part of the equality by maintaining the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$ and analogously for the less-than part. A detailed description of the algorithm for deriving a unary sum is provided in Appendix A.1.

If we want to encode a natural number j, for which we know that it can only take values in a small domain A, then introducing variables for all values in the range introduces a lot of redundant variables. 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. For a more concise encoding we can use a sparse representation, i.e., we represent $j \in \{0, 50, 75\}$ as $50 \cdot z_{50} + 25 \cdot z_{75}$ and enforce that $z_{50} \geq z_{75}$. In general, we use

$$\mathit{sparse}(\vec{z}, A) = \textstyle\sum_{i \in A \setminus \{0\}} (i - \mathit{pred}(i, A)) z_i \,, \tag{15}$$

where $\mathit{pred}(i, A) = \max(\{j \in A \mid j < i\})$. Additionally, we enforce that the z_i variables are ordered, i.e., $z_i \geq z_{\mathit{succ}(i,A)}$, where $\mathit{succ}(i, A) = \min(\{j \in A \cup \{\infty\} \mid j > i\})$. This representation is used in the sequential weight counter [31] and generalized totalizer encoding [32].

▶ **Proposition 6** (Sparse Unary Sum). *Given $A, B \subseteq \mathbb{N}$, $E = \{\, i + j \mid i \in A, j \in B \,\}$, ordering constraints on variables $\vec{y}$ and $\vec{y}'$, as well as fresh variables $\vec{z}$, we can derive*

$$\mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}', B) = \mathit{sparse}(\vec{z}, E)\,, \text{ and} \tag{16a}$$

$$z_i \geq z_{\mathit{succ}(i,E)} \quad i \in E \setminus \{\, \max(E) \,\}\,, \tag{16b}$$

using $O(|A| \cdot |B|)$ steps.

As in the case of the unary sum, these constraints can be added without changing satisfiability, because we can always set the fresh z_i variables such that the constraints are satisfied. The general idea is to introduce the fresh variables via reification $z_i \Leftrightarrow \sum_{i=1}^{n} \ell_i \geq i$. Then we simulate a brute-force search on the possible combinations of values for A and B, showing that the equality holds in all cases. A detailed description can be found in Appendix A.2.

Finally, if we want to represent a natural number that is large and has a large domain with maximal value m, then we can encode it using a binary representation, i.e., $j = \sum_{i=0}^{\lfloor \log_2(m) \rfloor} 2^i z_i$. To build a binary number (as is discussed in Section 5) it is sufficient to compose multiple full adders, which compute the sum of up to three input bits, using a binary adder circuit [18].

▶ **Proposition 7.** *For literals ℓ_1, ℓ_2, ℓ_3 and fresh variables z_1, z_0 we can derive the constraints*

$$\ell_1 + \ell_2 + \ell_3 = 2z_1 + z_0 \tag{17}$$

using $O(1)$ steps.

Again, it should be clear that this equality can be added without changing satisfiability because it can be satisfied using the fresh variables. To derive it, we reify

$$c \Leftrightarrow x + y + z \geq 2 \tag{18a}$$

$$s \Leftrightarrow x + y + z + 2\overline{c} \geq 3\,. \tag{18b}$$

The equality can be derived by multiplying (18a) by 2, adding (18b) and dividing the result by 3 as discussed in detail in [27].

In Section 5 and Appendix B, it is demonstrated how to apply this framework for the binary adder and the (generalized) totalizer encoding, respectively.

5 Binary Adder Encoding

The *binary adder encoding* [18] is used to encode general pseudo-Boolean constraints of the form $\sum_i a_i \ell_i \bowtie k$. The idea is to use an adder network to obtain the value of $\sum_i a_i \ell_i$ as a binary number $\sum_{i=0}^{\mathit{bits}} 2^i o_i$, where o_i are the output literals and $\mathit{bits} = \lfloor \log_2(\sum_i a_i) \rfloor$ is the required bit width. To enforce the constraint, the output bits o_i are constrained by clauses that perform a bitwise comparison with k in binary representation.

To recapitulate the algorithm for the construction of the adder network in [18], we need some more notation. A 2^m-*bit* is a literal that represents the numerical value 2^m. A 2^m-*bucket* is a queue of bits where each bit has the value 2^m and that supports operations to insert and extract bits. We use $[m]_2$ to denote the binary representation of a natural number m.

The construction of the network starts by initializing each 2^m-bucket with all literals ℓ_i such that the 2^m-bit of $[a_i]_2$ is 1. Then we repeat the following steps until there is at most one element left in each bucket. Consider the 2^m-bucket with the smallest value that has

Algorithm 2 Construction of adder network [18]. Procedure full_adder adds full adder to network.

```
1:  procedure adder_network(b)
2:      ▷ input: vector of buckets b
3:      for i from 0 to b.size() do
4:          while b_i.size()≥ 2 do
5:              if b_i.size()= 2 then
6:                  x, y ← b_i.dequeue()
7:                  c, s ← full_adder(x, y, 0)
8:              else
9:                  x, y, z ← b_i.dequeue()
10:                 c, s ← full_adder(x, y, z)
11:             b_i.enqueue(s)
12:             b_{i+1}.enqueue(c)
```

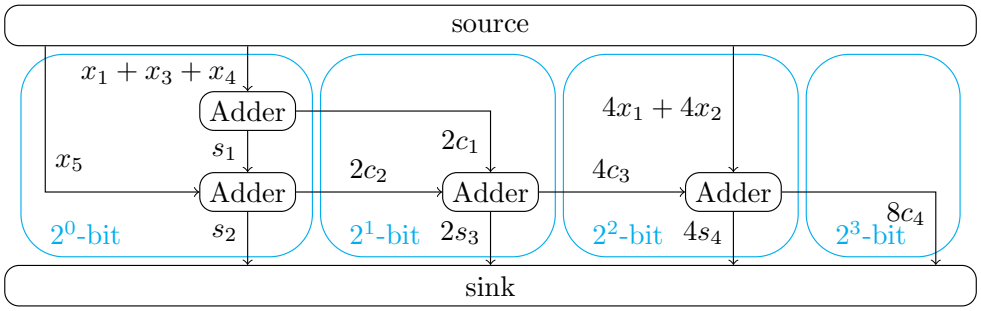


Figure 4 Layout of the arithmetic graph for adder network encoding of $5x_1+4x_2+x_3+x_4+x_5 \geq 5$.

at least 2 elements in it. If there are only 2 elements in the 2^m-bucket, take x and y from the bucket and set $z = 0$. Otherwise, let x, y and z be 3 elements from the 2^m-bucket and remove them from the 2^m-bucket. The bits x, y and z are used as input for a new full adder with fresh variables c and s as output, where c is a 2^{m+1}-bit and s is a 2^m-bit. The bits c and s are then inserted in their respective buckets, possibly creating a new bucket. An algorithm for constructing the network is given in Algorithm 2.

The arithmetic graph is constructed directly from the adder network such that each full adder is represented by a node. Each inner node constructed from the 2^m-bucket, i.e., which has 2^m-bits as input, has input edges with labels $2^m x$, $2^m y$ and $2^m z$ and output edges with labels $2^m s$ and $2^{m+1} c$. An example of the resulting graph is shown in Figure 4. The preserving equality can be derived using Proposition 7 and multiplying the resulting equality $x+y+z = 2c+s$ by 2^m to obtain $2^m x+2^m y+2^m z = 2^{m+1}c+2^m s$. After construction of the adder network, 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 $\sum_{i=0}^{bits} 2^i o_i$. If the 2^i-bucket is empty, o_i is set to 0.

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

$$
\begin{array}{cccc}
 & \overline{x}+\overline{y}+\overline{z}+s \geq 1 & & x+y+z+\overline{s} \geq 1 \\
\overline{y}+\overline{z}+c \geq 1 & \overline{x}+y+z+s \geq 1 & y+z+\overline{c} \geq 1 & x+\overline{y}+\overline{z}+\overline{s} \geq 1 \\
\overline{x}+\overline{z}+c \geq 1 & x+\overline{y}+z+s \geq 1 & x+z+\overline{c} \geq 1 & \overline{x}+y+\overline{z}+\overline{s} \geq 1 \\
\overline{x}+\overline{y}+c \geq 1 & x+y+\overline{z}+s \geq 1 & x+y+\overline{c} \geq 1 & \overline{x}+\overline{y}+z+\overline{s} \geq 1\,. \qquad (19)
\end{array}
$$

Note that all the clauses in (19) are RUP with respect to the preserving equality $x+y+z =$

$2c + s$.

To compare k with the output of the circuit, the encoding performs the comparison $\vec{x} \geq \vec{y}$ for bit vectors $\vec{x}$ and $\vec{y}$, where either $\vec{x} = o_{bits} \dots o_1 o_0$ 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. If we want to encode $\sum_{i=1}^{n} a_i \ell_i = k$, then the comparison for both directions is performed. If the size of these vectors is different, the shorter vector is padded with 0. Then, for $i = 0, \dots, bits$, the constraint

$$\overline{x}_i + y_i + \sum_{j=i}^{bits} x_j \overline{y}_j + \overline{x}_j y_j \geq 1 \tag{20}$$

is added to the CNF encoding. Note that either $\vec{x}$ or $\vec{y}$ is constant and hence the constraint is always a clause. This clause guarantees that the 2^i-bit on the variable side is equal to the 2^i-bit in $[k]_2$ or there was already a 2^j-bit for $j > i$ that is different to the 2^j-bit in $[k]_2$.

The clauses (20) are RUP with respect to $\sum_{i=0}^{bits} 2^i o_i \bowtie k$, which we obtain from the arithmetic graph using Proposition 4. The clauses are RUP because the RUP step will set all 2^j-bits, where $j > i$, to the same value as in $[k]_2$ and the 2^i-bit to the opposite value of the 2^i-bit in $[m]_2$, which falsifies $\sum_{i=0}^{bits} 2^i o_i \bowtie k$.

6 Experimental Results

To show the generality of our approach for proof logging arithmetic encodings, we implemented the sequential counter encoding [46], binary adder encoding [18], totalizer [3] and generalized totalizer encodings [32], in a certified encoding framework called VeritasPBLib. This framework inputs a pseudo-Boolean formula in OPB format and returns a CNF translation with the corresponding proof logging certificate. We used the verifier *VeriPB* [48] to verify the proof logging certificate returned by VeritasPBLib. The CNF formula is then solved by a modified version of the SAT solver *kissat* [34] [1] that generates proof logging compatible with the *VeriPB* verifier. Finally, we conjoin the proof logging from the CNF translation with the proof logging from SAT solving and verify the end-to-end pipeline with *VeriPB*.

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, since verification is slower than solving the problem.

To evaluate VeritasPBLib, we collected 1,803 pseudo-Boolean formulas from the PB 2016 Evaluation.[2] We can split these instances into four categories: (1) formulas with only clauses (279 instances), (2) formulas with clauses and cardinality constraints (772 instances), (3) formulas with clauses and general PB constraints (444 instances), and (4) formulas with clauses, cardinality and general PB constraints (308 instances). Since this work targets the verification of formulas with cardinality or general PB constraints, we excluded the 279 pure CNF formula instances, as those can already be certified with existing techniques. More details about the instances can be found in Appendix C.1.

The goal of our evaluation is to answer the following questions:

[1] Available at `https://gitlab.com/MIAOresearch/kissat_fork`
[2] Available at `http://www.cril.univ-artois.fr/PB16/`

Table 1 Number of translated, solved and verified instances for each encoding

Category	#Inst	Encoding	Translation		Solving			
			#CNF	#Veri	#Solved		#Verified	
					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

1. Can we use the end-to-end framework to verify the results of SAT-based approaches to solve pseudo-Boolean formulas and how efficient is verification?
2. How long does verification of the proof logging take when compared to translating the pseudo-Boolean formula to CNF?

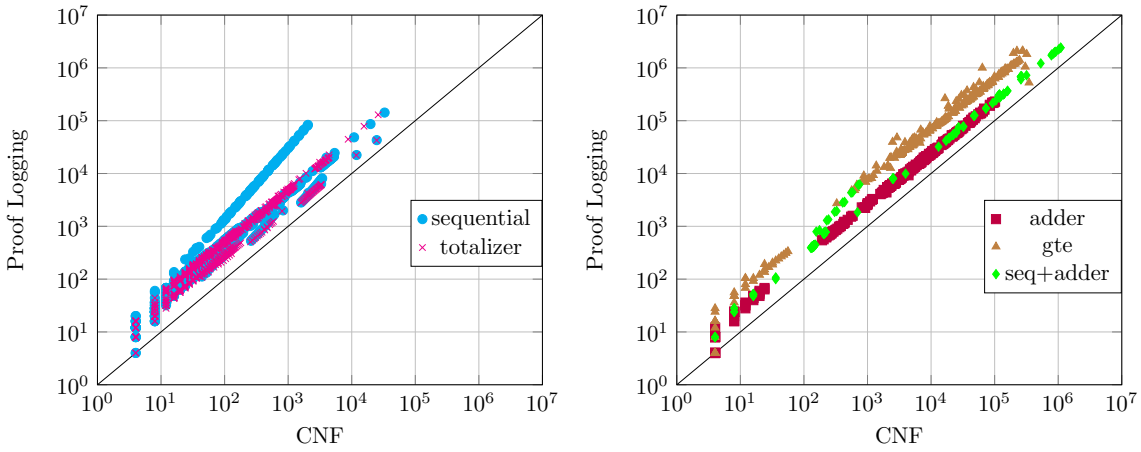
End-to-End Solving and Verification Table 1 shows how VeritasPBLib can be used to generate a CNF formula that can be solved by *kissat* and verified by *VeriPB*. For instances with cardinality constraints (*Card*), we use the sequential and totalizer encoding to translate those constraints to CNF. For instances with general PB constraints (*PB*), we use the adder and generalized totalizer encoding (GTE) to translate general PB constraints to CNF. Finally, for instances with both cardinality and general PB constraints (*Card+PB*), we use the sequential encoding for cardinality constraints and the adder 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 encodings but to show that we can verify the final result with a variety of encodings.

The column *#CNF* shows for how many instances VeritasPBLib successfully generated the CNF translation. For most of the formulas, we can translate the PB formula to CNF. The exceptions are 19 instances using the generalized totalizer (*GTE*) encoding 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 how many instances *VeriPB* can verify the proof logging certificate generated by VeritasPBLib. Except for a few instances for the *GTE* and *Seq+Adder* where the proof is large, *VeriPB* can verify the CNF translation. Note that if verification of the translation is successful, then this guarantees that the CNF encoding does not remove any solutions of the PB formula.

The columns *#Solved* and *#Verified* under *Solving* show how many instances can be solved by the SAT solver *kissat* and from those how many can be verified by *VeriPB*. If a satisfiable formula is verified, then it means that all clauses derived by *kissat* are due to correct derivations and the satisfying assignment returned by the SAT solver is a satisfying assignment of the original PB formula. If an unsatisfiable formula is verified, then it means that the reason of unsatisfiability is due to correct derivations.

We can verify 99% of the solved instances for unsatisfiable instances, which shows that the current approach can be used in practice to verify unsatisfiable results of SAT solvers when solving PB formulas. For satisfiable instances, we can verify 95% of the solved instances. However, for instances that *VeriPB* does not verify the result within the time limit, we can still certify that the satisfying assignment of the SAT solver satisfies the original PB formula. Even though *VeriPB* is already able to verify the majority of the proof logging,



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

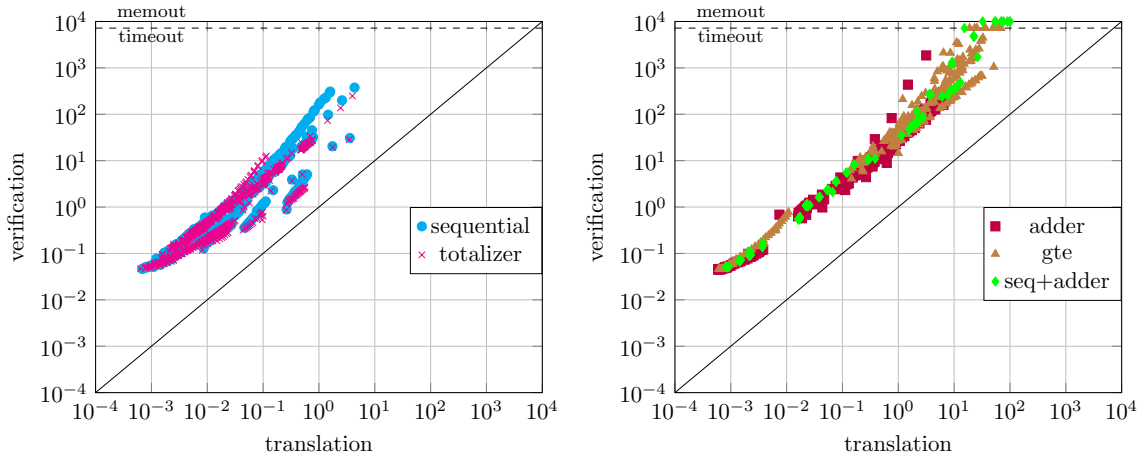
Figure 5 Comparison between CNF file size and proof logging file size in KiB

improvements to the verifier are orthogonal to our approach and can further increase the number of verified instances.

Translation and Verification Let us now focus on the CNF translation without solving. Our experiments show that the average overhead for proof logging ranges from 2× to 3× slower for all encodings with the exception of GTE which is around 5× slower. However, since translation is fast for the majority of instances (see Figure 6), the additional overhead of proof logging is not an issue when translating the PB formulas to CNF. A more detailed comparison of running times between CNF translation with and without proof logging can be found in Appendix C.2.

The overhead for translation can be explained with the increased proof size compared to the size of the CNF encoding as shown in Figure 5. For most instances the proof size seems to be within a constant factor of the CNF file size. However, there is a series of benchmarks for which the sequential counter encoding requires super linear (but still polynomial) proofs. It turns out that these instances are all crafted instances encoding a vertex cover [20]. These instances contain a constraint enforcing a constant fraction of the literals in the formula to be true, which is the worst case scenario for the sequential counter. At first glance, this super linear relationship seems to contradict the expected linear relationship between the number of clauses in the CNF and the number of steps in the proof. However, this can be explained as each reification step for deriving the unary sum introduces a constraint of linear size, so even though the number of steps for deriving a unary sum is linear, the proof size will be quadratic. It would be desirable to find a derivation of the unary sum that only requires linear proof size.

Figure 6 shows the relationship between the time to generate the CNF translation using VeritasPBLib and the time to verify the translation using *VeriPB*. The time to verify the translation compared to the translation itself is not negligible. Over all encodings, for 75% of benchmarks verification takes at-most 49 times longer than translation and for 98% of benchmarks take at-most 100 times longer. To some degree, such an overhead in verification time of the translation is expected, as the translation does not need to reason about its steps and the verification needs to perform some reasoning to justify the correctness of the proof steps. However, this also indicates that there is still room for improvement, both in terms of improving the performance of the verifier but also finding easier to verify derivation steps.



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 6 Comparison between CNF translation and verification of the corresponding proof logging

7 Concluding Remarks

In this work, we develop a general framework for certified translations of pseudo-Boolean constraints into CNF using cutting-planes-based proof logging. Since our method is a strict extension of *DRAT*, the proof for the translation can be combined with a SAT solver *DRAT* proof log to provide, for the first time, end-to-end verification for CDCL-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, our experimental evaluation shows that this approach is feasible in practice.

While studying the different encodings, we discovered the following interesting questions, which, to the best of our knowledge, have not been investigated before. For encodings using the order encoding, what impact does including the ordering constraints in the CNF translation have on the SAT solving? What is the effect on SAT solving for encoding $\sum_{i=1}^{n} a_i \overline{\ell_i} \leq \sum_{i=1}^{n} a_i - k$ instead of $\sum_{i=1}^{n} a_i \ell_i \geq k$ depending on the smaller degree? Does encoding the left-hand side of equality constraints only once instead of once for each direction have an impact on SAT solving time, especially with respect to propagation?

In our view, proof logging for pseudo-Boolean decision problems is only a first step. We believe that our method should also be sufficient to support proof logging for MaxSAT solvers. As a concrete example, using the techniques developed in this paper it should be possible to certify the clauses added during core extraction and objective function reformulation in *core-guided MaxSAT solving* [23, 38]. While supporting MaxSAT solvers using approaches such as *implicit hitting set (IHS)* [17] and *abstract cores* [6] seems a bit more challenging, we are still hopeful that our work could lead to a unified proof logging method for both MaxSAT solving and pseudo-Boolean optimization using cutting-planes-based reasoning as in [22, 35].

References

1 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.

2 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, MarchApril 2021.

3 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.

4 Lee A. Barnett and Armin Biere. Non-clausal redundancy properties. In *Proceedings of the 28th International Conference on Automated Deduction (CADE-28)*, 2021.

5 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.

6 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.

7 Armin Biere. Tracecheck. `http://fmv.jku.at/tracecheck/`, 2006. Accessed on 2021–03–19.

8 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.

9 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.

10 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)*, April 2022. To appear.

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

12 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.

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

14 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.

15 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.

16 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.

17 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.

18 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.

19 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.

20 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.

21 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.

22 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.

23 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.

24 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.

25 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.

26 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.

27 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.

28 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.

29 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.

30 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.

31 Steffen Hölldobler, Norbert Manthey, and Peter Steinke. A compact encoding of pseudo-boolean constraints into SAT. In Birte Glimm and Antonio Krüger, editors, *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.

32 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.

33 Daniela Kaufmann, Mathias Fleury, and Armin Biere. The proof checkers pacheck and pastèque for the practical algebraic calculus. In *Proceedings of the 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD '20)*, pages 264–269. IEEE, 2020.

34 Kissat SAT solver. `http://fmv.jku.at/kissat/`.

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

36 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*.

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

38 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.

39 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.

40 NaPS (Nagoya pseudo-Boolean solver). `https://www.trs.cm.is.nagoya-u.ac.jp/projects/NaPS/`.

41 Open-WBO: An open source version of the MaxSAT solver WBO. `http://sat.inesc-id.pt/open-wbo/`.

42 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.

43 Pseudo-Boolean competition 2016. `http://www.cril.univ-artois.fr/PB16/`, July 2016.

44 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.

45 The international SAT Competitions web page. `http://www.satcompetition.org`.

46 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.

47 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`.

48 VeriPB: Verifier for pseudo-Boolean proofs. `https://gitlab.com/MIAOresearch/VeriPB`.

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

50 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 Internatjuional Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, July 2014.

A Derivations for Building Blocks

Before going into detail on the derivations and presenting their respective algorithms, the notation for the proof logging is described. This is similar to the notation of the proof file used by *VeriPB*.

Algorithm 3 Deriving a unary sum over fresh variables z_i.

procedure derive_unary_sum(C')
  ▷ input: C' is of the form $\sum_{i=1}^{n} \ell_i = \sum_{i=1}^{n} z_i$ and describes 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 from 1 to k **do**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow$ reify($z_j \Leftrightarrow \sum_{i=1}^{n} 1 \cdot \ell_i \geq j$) ▷ Step 1: introduce variables as reification
  $C^{\text{geq}} \leftarrow$ derive_sum($D_1^{\text{geq}}, D_2^{\text{geq}}, \ldots, D_n^{\text{geq}}$) ▷ Step 2: derive $\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i$
  $C^{\text{leq}} \leftarrow$ derive_sum($D_n^{\text{leq}}, D_{n-1}^{\text{leq}}, \ldots, D_1^{\text{leq}}$) ▷ Step 3: derive $\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i$
  for i from 1 to $k-1$ **do**
    derive_ordering(D_i^{leq}, D_{i+1}^{geq}) ▷ Step 4: derive $z_i \geq z_{i+1}$, $i \in [n-1]$
  return $C^{\text{geq}}, C^{\text{leq}}$

Algorithm 4 Reify $\sum_{i=1}^{n} a_i \ell_i \geq j$ using the fresh variable z_j.

procedure reify($z_j \Leftrightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$)
  $C^{\text{geq}} \leftarrow \sum_{i=1}^{n} a_i \ell_i + j\bar{z}_j \geq j$ ▷ $z_j \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form
  proof_log(`red` C^{geq} `;` z_j `-> 0`)
  $C^{\text{leq}} \leftarrow \sum_{i=1}^{n} a_i \bar{\ell}_i + (\sum_{i=1}^{n} a_i - j + 1) z_j \geq \sum_{i=1}^{n} a_i - j + 1$ ▷ $z_j \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq j$ in normalized form
  proof_log(`red` C^{leq} `;` z_j `-> 1`)
  return $C^{\text{geq}}, C^{\text{leq}}$

Lines are added to the proof file using the proof_log(·) command. In this format, every constraint in the proof gets a unique *identifier* (or just *id* for brevity). We can express cutting planes derivations in reverse polish notation where constraints are referred to by their ids. For example, given previously derived constraints C and D, the line 'proof_log(`pol C D + 3 * 4 d`)' adds C and D, multiplies the result by 3, and finally divides by 4 (rounding up). In the concrete format constraints in reverse polish notation are represented by an identifier, but we omit this detail for simplicity and operate on the constraints directly. The proof format also supports the *saturation* rule, which, given a normalized constraint $\sum_i a_i \ell_i \geq A$, allows to derive $\sum_i \min(a_i, A)\ell_i \geq A$. We use 'proof_log(`pol C s`)' to denote saturation in the proof format.

A RUP constraint C can be added using 'proof_log(`rup` C)'. The syntax for adding a constraint as reification is 'red $z \Rightarrow C$; z 1' and 'red $z \Leftarrow C$; z 0', respectively (for more details please refer to [27]).

A.1 Deriving the Unary Sum

Deriving the constraints of a unary sum over fresh variables z_j, i.e.,

$$\sum_{i=1}^{n} \ell_i \geq \sum_{i=1}^{n} z_i \,, \tag{21a}$$
$$\sum_{i=1}^{n} \ell_i \leq \sum_{i=1}^{n} z_i \,, \text{ and} \tag{21b}$$
$$z_i \geq z_{i+1} \quad i \in [n-1] \,, \tag{21c}$$

is described in Algorithm 3, which is split into four steps. **Step 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 4 for the more general case using arbitrary positive coefficients.

Algorithm 5 Derive sum of reification variables.

procedure derive_sum($D_1, \ldots, D_n$)
  ▷ input: D_j is of the form $\sum_{i=1}^{n} \ell_i + j\overline{z}_j \geq j$
  $C \leftarrow D_1$
  for j from 2 to n **do** ▷ Invariant: $C : \sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$
    proof_log(`pol` C $j-1$ `*` D_j `+` j `d`)
    $C \leftarrow ((j-1) \cdot C + D_j)/j$
  return C

Algorithm 6 Deriving an ordering constraint $z_A \geq z_B$ from the reification constraints.

procedure derive_ordering(C, D)
  ▷ input: C is of form $z_A \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq A$
  ▷ input: D is of form $z_B \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq B$
  divisor $\leftarrow \sum_{i=1}^{n} a_i$
  ▷ derive $z_A \geq z_B$ if $A < B$
  proof_log(`pol` C D `+` *divisor* `d`)

Step 2: Deriving the Lower Bound. To derive (21a) in Algorithm 5 we maintain the invariant $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i \geq j$, which holds by induction. For $j = 1$ 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 the base case is covered. For the inductive step going from j to $j+1$, we multiply the invariant by j and add the reification constraint $z_{j+1} \Rightarrow \sum_{i=1}^{n} \ell_i \geq j+1$, which is $\sum_{i=1}^{n} \ell_i + (j+1)\overline{z}_{j+1} \geq j+1$ in normalized form, to get $(j+1)\sum_{i=1}^{n} \ell_i + j\sum_{i=1}^{j} \overline{z}_i + (j+1)\overline{z}_{j+1} \geq j^2 + j + 1$. Note that $j^2 + j + 1 = (j+1)^2 - j$ and hence division by $j+1$ and rounding up yields $\sum_{i=1}^{n} \ell_i + \sum_{i=1}^{j} \overline{z}_i + \overline{z}_{j+1} \geq j+1$, i.e., the invariant for $j+1$. For $j = k+1$ the invariant is the normalized form of (21a).

Step 3: Deriving the Upper Bound. To derive (21b) we can use Algorithm 5 again but need to provide the constraints in reverse order to fit the required input format.

Step 4: Deriving the Ordering Constraints. The ordering constraint is derived in Algorithm 6, using the reification constraints: We add the constraints used for reification, that is $z_{j+1} \Rightarrow \sum_{i=1}^{n} a_i \ell_i \geq j+1$ and $z_j \Leftarrow \sum_{i=1}^{n} a_i \ell_i \geq j$. 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, e.g., m.

A.2 Deriving the Sparse Unary Sum

In this section we prove Proposition 6 by providing Algorithm 7, which derives the sparse unary sum of two numbers in sparse unary representation. As for the unary sum, we start in Step 7.1 by introducing the required fresh variables via reification. However, we only need to introduce the variables that will be used, i.e., those 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 introduction of variables with index bigger than k can be avoided by having a arithmetic graph each for the upper and lower bound and relaxing the preserving equality to inequalities.) After introducing the variables we can derive the ordering constraints as before.

In Step 7.2 we introduce a variable z_{eq} which is true if and only if the equality to be derived is true. Note that we need to represent an equality as two inequalities and hence

need to introduce separate variables z_{geq}, z_{leq} for each inequality and then combine them into z_{eq}.

In Step 7.3 we derive $z_{eq} \geq 1$ by checking all combinations of values in A and B, which requires $O(|A| \cdot |B|)$ steps. Note that asymptotically this is the same number of steps as is required to compute which elements are in E so this step is still linear in the time needed to construct the encoding.

In Step 7.4 we use that $z_{eq} \geq 1$ and hence $z_{geq} = z_{leq} = 1$, which allows us to derive $sparse(\vec{y}, A) + sparse(\vec{y}', B) \geq sparse(\vec{z}, E)$ and $sparse(\vec{y}, A) + sparse(\vec{y}', B) \leq sparse(\vec{z}, E)$ respectively by removing z_{geq}, z_{leq} from the constraints introduced in Step 7.2.

Algorithm 8 describes in detail how to derive $z_{eq} \geq 1$ by checking all combinations of values in A and B. Let us illustrate how the algorithm works with an example. Let $A = \{0, 2\}$ and $B = \{0, 2, 4\}$. After the first iteration of the outer loop the algorithm derives the clauses

$$y_2 + y_2' + z_{eq} \geq 1\,, \tag{22a}$$
$$y_2 + \overline{y}_2' + y_4' + z_{eq} \geq 1\,, \text{ and} \tag{22b}$$
$$y_2 + \overline{y}_4' + z_{eq} \geq 1\,. \tag{22c}$$

Note that deriving (22a) by reverse unit propagation sets $y_2 = y_2' = z_{eq} = 0$. This causes the ordering constraints to propagate all variables in $\vec{y}$ and $\vec{y}'$. As all $\vec{y}$ and $\vec{y}'$ variables are set, the reification constraints introduced in Step 7.1 will cause all $\vec{z}$ variables to propagate. As the constraints reified in Step 7.2 are now satisfied we also get the propagation $z_{geq} = z_{leq} = 1$ and hence z_{eq} should be set to 1 as well. However, we already set z_{eq} to 0 and hence have a contradiction showing that (22a) can be derived. Deriving the other clauses works analogously.

If we add all clauses in (22) together, then y_2' and y_4' get canceled out and we are left with $3y_2 + 3z_{eq} \geq 1$ which is saturated to obtain $y_2 + z_{eq} \geq 1$. Analogously, the second iteration of the outer loop derives $\overline{y}_2 + z_{eq} \geq 1$, which added to the result of the first iteration yields $2z_{eq} \geq 1$ and using saturation we obtain $z_{eq} \geq 1$ as desired.

A.3 Derivation for binary adder encoding

This section provides the algorithm for proof logging and derivation of the preserving equality (17) from Proposition 7 for a single binary full adder in Algorithm 9.

B Totalizer and Generalized Totalizer Encoding

The totalizer and generalized totalizer encoding accumulate the input in form of a balanced binary tree. The totalizer encoding is designed for encoding cardinality constraints and uses the order encoding to represent values, while the generalized totalizer is designed for general pseudo-Boolean constraints and uses a sparse representation. An example of an arithmetic graph for the generalized totalizer encoding is shown in Figure 7. This graph contains a leaf node for each of the variables in the encoded constraint (to obtain a unique source we simply combine all leaf nodes into one node). The leaf nodes are combined in form of a binary tree, where we ensure that the value is preserved for each inner node, i.e., each possible value of incoming edges is representable as value of the outgoing edges. To perform k-simplification the arithmetic graph has additional edges that go directly into the sink node. The formal definition of arithmetic graph for the (generalized) totalizer encoding is as follows.

Algorithm 7 Deriving a sparse unary sum over fresh variables $\vec{z}$.

procedure derive_sparse_unary_sum(C')
  ▷ input: C' is of the form $\mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}', B) = \mathit{sparse}(\vec{z}, E)$ and describes the constraint to be derived such that $A, B \subseteq \mathbb{N}$, $E = \{\, i + j \mid i \in A, j \in B \,\}$ and $\vec{z}$ variables are fresh
  ▷ **Step 7.1**: introduce variables as reification and derive ordering
  for $j \in E \setminus \{\, 0 \,\}$ **do**
    $D_j^{\text{geq}}, D_j^{\text{leq}} \leftarrow \text{reify}(z_j \Leftrightarrow \mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}', B) \geq j)$
  for $i \in E \setminus \{\, 0, \max(E) \,\}$ **do**
    derive_ordering(D_i^{leq}, $D_{succ(i,E)}^{\text{geq}}$) ▷ derive $z_i \geq z_{succ(i,E)}$
  ▷ **Step 7.2**: : reify constraint to be derived
  $C^{\text{geq}}, \_ \leftarrow \text{reify}(z_{geq} \Leftrightarrow \mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}', B) \geq \mathit{sparse}(\vec{z}, E))$
  $C^{\text{leq}}, \_ \leftarrow \text{reify}(z_{leq} \Leftrightarrow \mathit{sparse}(\vec{y}, A) + \mathit{sparse}(\vec{y}', B) \leq \mathit{sparse}(\vec{z}, E))$
  $\text{reify}(z_{eq} \Leftrightarrow z_{geq} + z_{leq} \geq 2)$
  ▷ **Step 7.3**: derive that $z_{eq} \geq 1$
  try_all_values($\mathit{sparse}(\vec{y}, A), \mathit{sparse}(\vec{y}', B), z_{eq}$)
  ▷ **Step 7.4**: derive constraint to be derived from its reification
  $M \leftarrow \max(A) + \max(B)$ ▷ Coefficient so that reification variables get eliminated.
  $D \leftarrow z_{geq} \geq 1$
  proof_log(`rup` D)
  proof_log(`pol` C^{geq} D M `* +`)
  $C^{\text{geq}} \leftarrow C^{\text{geq}} + M \cdot D$
  $D \leftarrow z_{leq} \geq 1$
  proof_log(`rup` D)
  proof_log(`pol` C^{leq} D M `* +`)
  $C^{\text{leq}} \leftarrow C^{\text{leq}} + M \cdot D$
  return $C^{\text{geq}}, C^{\text{leq}}$

▶ **Definition 8** (Arithmetic graph for the generalized totalizer encoding). *Given a linear sum $\sum_i a_i x_i$ 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). In what follows we will consider r as an inner node. The edge (s_i, v) from the leave s_i is labeled with $a_i x_i$, which can be viewed as a sparse representation for values $\{\, 0, a_i \,\}$. For an inner node v with two incoming edges with labels $\mathit{sparse}(\vec{y}, A)$ and $\mathit{sparse}(\vec{y}', B)$, the outgoing edge e is labeled $\mathit{sparse}(\vec{z}, E)$, where $\vec{z}$ are fresh variables and $E = \{\, i + j \mid i \in A, j \in B \,\}$. To obtain a graph with a single source we combine all s_i into a single node s. To perform k-simplification we split $\mathit{sparse}(\vec{z}, E) = \sum_{i \in E} a_i z_i$ into $\sum_{i \leq succ(k,E)} a_i z_i$, which is the label of the outgoing edge e, and $\sum_{i > succ(k,E)} a_i c_i$, which is the label for an addition outgoing edge $e' = (v, t)$.*

To see that the defined graph is an arithmetic graph, we only need to check that we can derive the preserving equality for each inner node. Each inner node has two incoming edges that are labeled with a sparse unary representation and all outgoing edges together form a sparse unary representation as well, so that we can use Proposition 6 to derive the required preserving equality. Note that Proposition 6 also requires to have ordering constraints on the input variables, however, it is easy to see by an inductive argument that the ordering constraints on the variables will be present, when processing the graph in topological order: Edges from the source only contain a single variable and hence the ordering constraints exist

Algorithm 8 Given a reified sparse unary sum, derive that the reification variable is true.

procedure fix($sparse(\vec{y}, A), a$)
  return $\overline{y}_a + y_{succ(a,A)}$ ▷ replace y_0 by 1 and y_∞ by 0
procedure try_all_values($sparse(\vec{y}, A), sparse(\vec{y}', B), z_{eq}$)
  $C_{outer} \leftarrow 0 \geq 0$
  for $i \in A$ **do**
    $C_{inner} \leftarrow 0 \geq 0$
    for $j \in B$ **do**
      ▷ assuming that a (respectively b) is the value encoded by $sparse(\vec{y}, A)$ ($sparse(\vec{y}', B)$)
      ▷ encode that $(a = i \wedge b = j) \Rightarrow z_{eq}$
      $D \leftarrow fix(sparse(\vec{y}, A), i) + fix(sparse(\vec{y}', B), j) + z_{eq} \geq 1$
      proof_log(`rup` D)
      proof_log(`pol` C_{inner} D `+`)
      $C_{inner} \leftarrow C_{inner} + D$
    proof_log(`pol` C_{outer} C_{inner} `s +`)
    $C_{outer} \leftarrow C_{outer} + \mathsf{saturate}(C_{inner})$
  proof_log(`pol` C_{outer} `s`)
  $C_{outer} \leftarrow \mathsf{saturate}(C_{outer})$
  return C_{outer} ▷ C_{outer} is now $z_{eq} \geq 1$

Algorithm 9 Proof logging the encoding of a single full adder.

procedure full_adder(x, y, z)
  $D^{\text{geq}}_{carry}, D^{\text{leq}}_{carry} \leftarrow \text{reify}(c \Leftrightarrow x + y + z \geq 2)$
  $D^{\text{geq}}_{sum}, D^{\text{leq}}_{sum} \leftarrow \text{reify}(s \Leftrightarrow x + y + z + 2\overline{c} \geq 3)$
  $D^{\text{geq}} \leftarrow (2 \cdot D^{\text{geq}}_{carry} + D^{\text{geq}}_{sum})/3$
  proof_log(`pol` D^{geq}_{carry} `2 *` D^{geq}_{sum} `+ 3 d`)
  $D^{\text{leq}} \leftarrow (2 \cdot D^{\text{leq}}_{carry} + D^{\text{leq}}_{sum})/3$
  proof_log(`pol` D^{leq}_{carry} `2 *` D^{leq}_{sum} `+ 3 d`)
  return $D^{\text{geq}}, D^{\text{leq}}, c, s$ ▷ D is the preserving equality of the full adder

trivially. For inner nodes we get the ordering constraints by applying Proposition 6.

If the set of achievable values E is dense for some node, i.e., E contains all values from 0 to $\max(E)$, then we can also use Proposition 5 to derive the required preserving equality, which only requires $O(|E|)$ instead of $O(|A| \cdot |B|)$ steps and hence can reduce the proof logging overhead.

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

$$\overline{y}_i + \overline{y}'_j + z_{i+j} \geq 1 \quad \text{for } i \in A, j \in B \tag{23a}$$

$$y_{succ(i,A)} + y'_{succ(j,B)} + \overline{z}_{succ(i+j,E)} \geq 1 \quad \text{for } i \in A, j \in B \text{ s.t. } i + j \tag{23b}$$

where $succ(i, A) = \min(\{\, j \mid j \in A \cup \{\infty\}, j > i \,\})$ and we replace y_0, y'_0 with 1, and $y_\infty, y'_\infty, z_\infty$ with 0 and simplify accordingly. Note that, (23) encodes that $a + b = c$ (where a and b are the incoming values and c is the output value), because (23a) encodes that if $a \geq i$ (expressed by assigning y_i to 1) and $b \geq j$ then $c \geq i + j$ while (23a) encodes that if $a \leq i$ (which is the same as saying that $a < succ(i, A)$, expressed by assigning $y_{succ(i,A)}$ to 0) and $b \leq j$ then $c \leq i + j$.

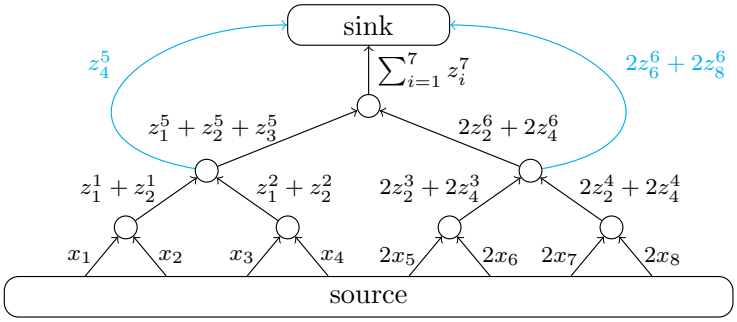


Figure 7 Layout of the arithmetic graph for the generalized totalizer encoding of $x_1 + x_2 + x_3 + x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \leq 2$. Edges introduced for k-simplification are colored cyan.

Table 2 Properties of pseudo-Boolean formulas used in the experimental results.

		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

For proof logging the CNF encoding we can simply add all clauses using RUP: A RUP check of (23a) will assign $y_i = y'_j = 1$ and $z_{i+j} = 0$. The ordering constraints on $\vec{y}, \vec{y}'$ will cause a propagation setting multiple $\vec{y}, \vec{y}'$ variables to true such that $\mathit{sparse}(y, A) + \mathit{sparse}(y', B)$ has a value of at least $i + j$, while the ordering constraints on $\vec{z}$ will propagate multiple $\vec{z}$ to false such that $\mathit{sparse}(z, E)$ can only take a value that is strictly less than $i + j$ and hence causes a conflict with the preserving equality $\mathit{sparse}(z, E) = \mathit{sparse}(y, A) + \mathit{sparse}(y', B)$. Similarly, a RUP check of (23b) will assign $y_{\mathit{succ}(i,A)} = y'_{\mathit{succ}(j,B)} = 0$ and $z_{\mathit{succ}(i+j,E)} = 1$ causing propagations such that $\mathit{sparse}(y, A) + \mathit{sparse}(y', B)$ takes a value less than or equal to $i + j$ and $\mathit{sparse}(z, E)$ takes a value strictly greater than $i + j$ causing again a conflict with the preserving equality.

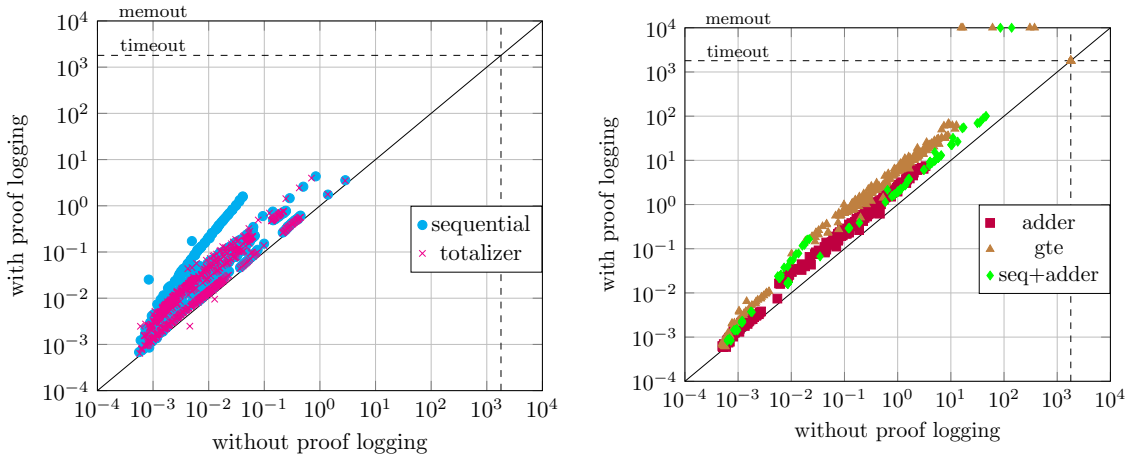
To enforce a pseudo-Boolean constraint $\sum_i a_i x_i \bowtie k$, we first derive a bound on the output of the arithmetic graph $\sum_i c_i o_i \bowtie k$, using Proposition 4. Then we can derive unit clauses on the output via reverse unit propagation.

To encode $\sum_i a_i x_i \geq k$ or $\sum_i a_i x_i \leq k$ the clause $z_{\mathit{succ}(k-1,E)} \geq 1$ or $\overline{z}_{\mathit{succ}(k,E)} \geq 1$ 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 $\sum_i a_i x_i = k$ both clauses are added.

C Additional Evaluation Data

C.1 Benchmarks

Table 2 shows some properties of the benchmarks used in the experimental results, namely, the average number of cardinality constraints (Card), the average number of literals in each constraint, and the average size of coefficients associated with each literal. (The same is



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 8 Comparison of runtimes between CNF translation with and without proof logging.

shown for PB constraints.) Since the benchmark set is composed of instances from multiple domains, there is a large dispersion 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 have an even larger dispersion with instances that have from 1 to 2,378,901 PB constraints and from 1 to 75,582 cardinality constraints.

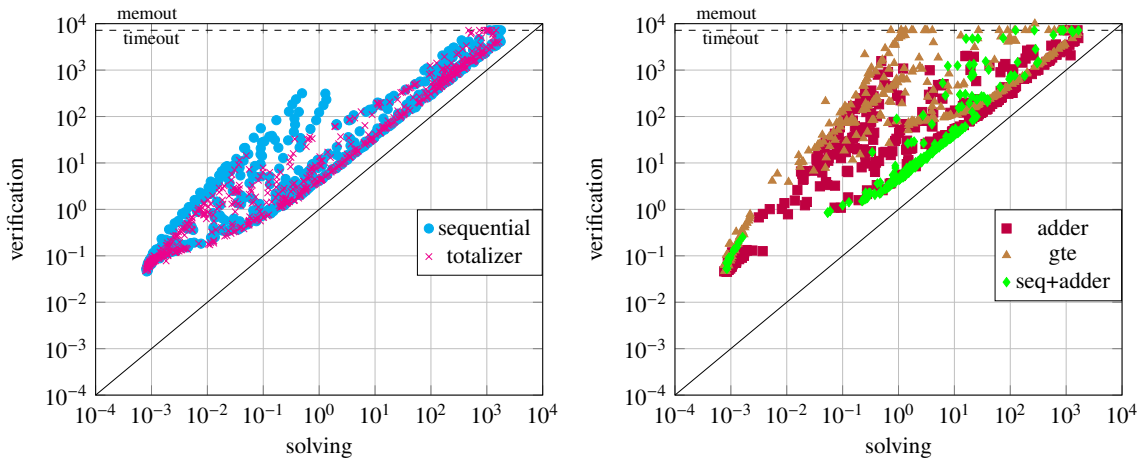
C.2 Overhead of Proof Logging

Figure 8 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 exception are the cardinality formulas from vertex cover that require super linear proofs, which lead to a higher overhead when storing the proof. Additionally, there were 6 instances that had memory outs when storing the proof in memory, which could be improved in the future by a more compact representation of the proof logging in VeritasPBLib.

C.3 Solving and Verification

Figure 9 shows the relationship between the time to generate the CNF translation and solve it using *kissat* and the time to verify the translation and solution using *VeriPB*. It can be seen that even though we can verify most instances, verification is often considerably slower than solving.

A lot of instances are spread in a wide range of different overheads. This wide range only comes from verifying the solution, which is out of the scope of this work. However, it motivates potential improvements to *VeriPB* which are complementary to the work proposed in this paper and can further increase the number of verified instances.



(a) Cardinality formulas

(b) General pseudo-Boolean formulas

Figure 9 Comparison between end-to-end solving and verification time

Paper E

Justifying All Differences Using Pseudo-Boolean Reasoning

Jan Elffers,[1,2] **Stephan Gocht,**[1,2] **Ciaran McCreesh,**[3] **Jakob Nordström**[2,4]
[1]Lund University, Lund, Sweden
[2]University of Copenhagen, Copenhagen, Denmark
[3]University of Glasgow, Glasgow, Scotland
[4]KTH Royal Institute of Technology, Stockholm, Sweden
{jan.elffers, stephan.gocht}@cs.lth.se, ciaran.mccreesh@glasgow.ac.uk, jn@di.ku.dk

Abstract

Constraint programming solvers support rich global constraints and propagators, which make them both powerful and hard to debug. In the Boolean satisfiability community, proof-logging is the standard solution for generating trustworthy outputs, and this has become key to the social acceptability of computer-generated proofs. However, reusing this technology for constraint programming requires either much weaker propagation, or an impractical blowup in proof length. This paper demonstrates that simple, clean, and efficient proof logging is still possible for the all-different constraint, through pseudo-Boolean reasoning. We explain how such proofs can be expressed and verified mechanistically, describe an implementation, and discuss the broader implications for proof logging in constraint programming.

Introduction

Constraint programming solvers are increasingly being used for fully automated decision making without a human in the loop, even in safety-critical applications. Unfortunately, these solvers will sometimes have bugs, and these bugs are hard to detect using conventional testing methods (Akgün et al. 2018; Gillard, Schaus, and Deville 2019). Meanwhile, formal proofs of correctness can be useful in verifying the mathematical description of some of the algorithms underlying these solvers, but are not yet suitable for verifying a full implementation of a high-performance modern solver. It would therefore be reassuring to have a different way to be confident that a solver has produced a correct answer.

When a constraint programming solver outputs "yes" for a decision instance, it is usually relatively easy to verify that the answer it provides is valid—for example, by having a different person implement a solution checker, which is typically much simpler than writing a program which finds a solution. Similarly, for optimisation problems, verifying the *feasibility* of a solution is simple. However, for "no" decision instances, and for verifying optimality, a solver likely took a large number of complicated steps to reach that conclusion, and there is no simple way of demonstrating that those steps were valid. In the Boolean satisfiability community, proof logging is the standard approach to this problem: in order to take part in the SAT competitions (Heule, Järvisalo, and Suda 2019), a solver must be able to output a "certificate" or "proof log" alongside a claimed unsatisfiable result. This log is a (potentially very large) file in a standard format known as DRAT (Heule, Hunt Jr., and Wetzler 2013b; 2013a; Wetzler, Heule, and Hunt Jr. 2014), which may be verified by an external tool. Importantly, the proof verifying tools used are much simpler than the solvers, giving us confidence in their correctness. Proof logging for Boolean satisfiability has been key to the social acceptance of computer-produced proofs of mathematical conjectures (e.g. Heule, Kullmann, and Marek 2016, Lamb 2016).

Due to their use of rich global constraints like "all different", constraint programming solvers cannot simply reuse this approach: compiling constraint programming to Boolean satisfiability results in weaker reasoning for many constraints (Bessiere et al. 2009b), and although it is *theoretically* possible to use the DRAT format to justify rich propagation, developing any approach that is feasible *in practice* has remained stubbornly out of reach.

In this work, we instead propose the use of a new proof format based upon pseudo-Boolean reasoning and cutting planes proofs (Cook, Coullard, and Turán 1987). We show that this format can easily and efficiently capture all of the reasoning carried out by the all different propagator. This allowing us to develop, for the first time, an efficient verification system for non-trivial constraint programming inference techniques: we describe a tool which can verify these proofs, as well as the implementation of a small constraint solver which produces them. We conclude with a discussion of the broader implications for constraint programming.

Pseudo-Boolean Models

An instance of the pseudo-Boolean (PB) decision problem, or a PB formula, is defined by a set of $\{0,1\}$-valued variables $\{x_1,\dots,x_n\}$ and a set of linear constraints over these variables, each of which is of the form $\sum_{i=1}^{n} a_i\ell_i \geq A$, where the a_is and A are all integers, and each ℓ_i is either an unnegated literal x_i or a negated literal $\overline{x_i}$. Using the equality $x_i + \overline{x_i} = 1$, which encodes the semantics of negation, we can always rewrite a PB constraint so that all a_i are non-

negative and A is strictly positive, and so when describing reasoning rules we will assume that all constraints are written in this so-called *normalized form* (this is purely for notational convenience, and does not affect expressive power). For a constraint in normalized form, A is often referred to as the *degree of falsity*, or just *degree*. The objective is to assign values to the variables so that all constraints are respected. If this is possible, we say that the PB instance is *satisfiable*; otherwise it is *unsatisfiable*.

By treating 0 as "false" and 1 as "true", any instance of the Boolean satisfiability (SAT) problem in conjunctive normal form (CNF) can be viewed as a PB formula by observing that, e.g. $x \vee \overline{y} \vee z$ is satisfied if and only if $x + \overline{y} + z \geq 1$. On the other hand, not all pseudo-Boolean constraints can be translated into a single SAT clause. For example, "cardinality" constraints such as $x_1 + x_2 + x_3 + x_4 + x_5 \geq 3$ must be encoded before they can be handled by a SAT solver, and more general constraints such as $x_1 + 2\overline{x_2} + 3x_3 + 4\overline{x_4} + 5\overline{x_5} \geq 7$ require even more complicated handling.

Cutting Planes Proofs

To reason about satisfiability or unsatisfiability of pseudo-Boolean formulae we use the *cutting planes* proof system (Cook, Coullard, and Turán 1987), which can be described as follows. We have four sets of derivation rules, which we describe using the standard notation with a list of preconditions above a horizontal line that allow us to infer the constraint below the line. Initially, these preconditions can be the constraints in the PB formula, which we refer to as *(input) axioms*, but later they can be any constraint derived by a previous rule application. Firstly, we may assert unconditionally the *literal axiom* that any x_i or $\overline{x_i}$ is non-negative:

$$\frac{}{\ell_i \geq 0}$$

Secondly, we may create a new constraint by *addition* of any two constraints:

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

(Here we implicitly assume that the equality $x_i + \overline{x_i} = 1$ is applied to cancel any literals of opposing signs, and shift any constant terms to the right-hand side, so that the resulting constraint is in normalized form). Thirdly, we can apply *multiplication* by a positive integer c to any constraint:

$$\frac{\sum_i a_i\ell_i \geq A}{\sum_i ca_i\ell_i \geq cA}$$

Fourthly, we may apply *division* by any positive integer c, where all fractional values in the divided constraint are rounded upwards:

$$\frac{\sum_i a_i\ell_i \geq A}{\sum_i \lceil \frac{a_i}{c} \rceil \ell_i \geq \lceil \frac{A}{c} \rceil}$$

(Note that the soundness of this rule requires that the constraint is in normalized form.)

Cutting planes is a complete proof system for pseudo-Boolean formulas in the same way that the resolution proof system (Blake 1937; Davis and Putnam 1960) is complete for CNF formulas—it is always possible to derive $0 \geq 1$ from a PB formula using a cutting planes proof if and only if this formula is unsatisfiable. We refer the interested reader to Buss and Nordström (2019) for more details.

Unit Propagation

Two key notions in the context of SAT solving and pseudo-Boolean solving, which will be important for us also, are those of *propagating* and *conflicting* constraints. Let $C = \sum_i a_i\ell_i \geq A$ be a PB constraint and let ρ be a partial truth value assignment. Then the *slack* of C under ρ measures how much room there is left for error if we want to satisfy C given ρ. Formally, $\mathrm{slack}(C;\rho) = \sum_{i\,:\,\rho(\ell_i)\neq 0} a_i - A$ is the sum of the coefficients of all non-falsified literals minus the degree. If $\mathrm{slack}(C;\rho) < 0$, then there is no way C can be satisifed, and we say that the constraint is *conflicting* under ρ. If for some coefficient a_i we have $\mathrm{slack}(C;\rho) < a_i$, then ℓ_i must be set to true to avoid conflict, and we say that C *unit propagates* ℓ_i under ρ. By way of example, for the empty assignment the constraint $C = x_1 + 2\overline{x_2} + 3x_3 + 4\overline{x_4} + 5\overline{x_5} \geq 7$ has slack 8 and does not propagate anything, but if we set $x_5 = 1$ then the slack drops to 3 and C propagates $x_4 = 0$. If we instead set $x_4 = 1$, then the slack decreases to -1 and we have a conflict. We note that the pseudo-Boolean notation of unit propagation is just a generalization of that used in conflict-driven clause learning (CDCL, Marques Silva and Sakallah 1999), since a disjunctive clause unit propagates only when the slack is 0 (since all coefficients are 1), which happens precisely when all literals in the clause except one is falsified.

Reverse Unit Propagation

The concept of unit propagation has turned out to be very useful for proof logging as explained next. A constraint C can be derived from a PB formula F if and only if F together with the negation of C is unsatisfiable. In general, deciding whether this is so is an NP-complete problem, but in the context of CDCL solving it is much easier. Namely, if F is the set of clauses derived so far and C is the new clause learned from the most recent conflict, then it holds that F plus $\neg C$ (i.e., the conjunction of the negations of all literals in C) unit propagates to conflict. When this is the case, we say that C follows from F by *reverse unit propagation (RUP)* or is a *RUP clause*. The correctness of a basic CDCL proof search loop can be verified efficiently by just emitting the learned clauses one by one and checking that they are RUP clauses (Goldberg and Novikov 2003; Van Gelder 2008). The more expressive DRAT proof logging format (Heule, Hunt Jr., and Wetzler 2013b; 2013a; Wetzler, Heule, and Hunt Jr. 2014) used in current state-of-the-art SAT solvers is based on an extension of this simple but powerful idea.

The RUP concept readily transfers to a pseudo-Boolean setting. We say that the constraint $\sum_i a_i\ell_i \geq A$ is RUP for a PB formula F if the negation of this constraint (i.e. $\sum_i -a_i\ell_i \geq 1 - A$) together with F unit propagates to conflict, and if this is the case then it is clearly sound to derive

$\sum_i a_i \ell_i \geq A$ from F. This is useful in that, as we will show, it allows for very efficient proof logging for some constraint programming propagation algorithms.

Machine-Verifiable Proofs

In order to produce a machine-verifiable proof of unsatisfiability for a PB formula, we need a file that expresses the problem, and a second file that provides the proof. There is a standard format[1] for expressing pseudo-Boolean problems, which we use as a starting point. Briefly, each line in the file is either a comment (starting with an asterisk), or specifies a constraint. For example, the line

```
3 x1 + 2 ~x3 -3 x6 >= 2 ;
```

specifies that $3x_1 + 2\overline{x_3} - 3x_6 \geq 2$. For solver competitions, a series of additional guarantees are provided, such that the file will start with a special header comment, and that the variables will be named "x1" through "xN". Many pseudo-Boolean solvers treat these guarantees as requirements on the input, and will reject or misbehave if they are not followed, so these guarantees are *de facto* rules. Our tools *can* generate files following these restrictions, but do not have to do so: we find it more readable to be able to generate PB variable names like "xFoo_3" to correspond to a constraint programming variable "Foo" taking the value 3.

For logging proofs, we have created a new format. The format is a simple text file, which is at least somewhat human-readable, and which has been designed to reduce the amount of work required from solver implementers to a minimum. In particular, a key design choice is that solver writers will not need to maintain an entire pseudo-Boolean solver alongside their existing constraint programming solver, and can instead output proofs using a simpler template-based approach—we discuss this further in the following section.

Proof headers. The proof file must begin with a header line. Typically, this will immediately be followed by an "f" rule, as follows (the asterisk line is a comment and is ignored):

```
pseudo-Boolean proof version 1.0
* read in the 18 model constraints
f 18 0
```

This "f" rule instructs the proof verifier to read in the pseudo-Boolean model file. The "18" must correspond to the number of constraints in the problem, except that any "equals" constraint in the model is considered to be two inequalities instead. Each constraint read in is numbered, starting from 1. The zero is a line terminator.

Deriving constraints. Subsequent lines in a proof will use these numbered constraints, ultimately deriving a contradiction. The first way to do so is using a "p" rule, which takes an expression in reverse Polish notation, and creates a new numbered constraint with its result. For example, the line

```
p 42 3 * 43 + 2 d 0
```

means "create a new constraint by multiplying the constraint numbered 42 by 3, then adding constraint 43, then dividing by 2"; again, the zero is a line terminator. The "p" rule can thus express any number of applications of the addition, multiplication, and division axioms as a single step—during development we found this to be much more convenient and compact than requiring a step per axiom application.

Literal axioms. The "p" rule may also be used to introduce literal axioms. For example, the line

```
p x1 ~x2 + 5 + 0
```

will create a new constraint by adding the literal axioms $x_1 \geq 0$ and $\overline{x_2} \geq 0$ to constraint number 5.

Reverse unit propagation. The "u" rule gives another way of introduction of a new constraint, which this time is given explicitly in OPB format. For example,

```
u -1 x8 -1 x25 -1 x26 -1 x5 >= -3 ;
```

would create a new numbered constraint saying

$$-x_8 + -x_{25} + -x_{26} + -x_5 \geq -3.$$

In order for such a constraint to be introduced, it must be an "obvious" consequence of the constraints known so far. Here "obvious" is defined to mean "follows by reverse unit propagation", as described in the previous section.

Although the "u" rule is theoretically no more powerful than the "p" rule, using this rule substantially reduces the implementation effort for solver authors. It avoids the need for solvers to understand pseudo-Boolean constraints to (e.g.) perform cancellations correctly, and instead offloads that work onto the proof verifier. It also avoids the need to explicitly log any steps for propagation of constraints which can be encoded into pseudo-Boolean form in a way where unit propagation gives the same propagation strength as the constraint.

Asserting contradiction. Once a contradiction has been derived, the "c" rule is used to verify that assertion and terminate the proof. So, a typical proof may end as follows:

```
u >= 1 ;
c 146750 0
```

Here the penultimate line asserts that contradiction ($0 \geq 1$) follows by unit propagation from the constraints learned so far, and the final line asserts that the previous constraint (which has number 146750) is in fact a contradiction.

Other rules. The proof format also supports other rules, including ways of deleting constraints (for reduced memory usage) and verifying solutions. These are explained in the documentation for our proof-checking tool, which we will now describe.

[1] http://www.cril.univ-artois.fr/PB12/format.pdf

A Proof Checking Tool

We have implemented a proof checking tool for this proof format.[2] It is written in Python, with critical parts in C++ for performance reasons. The tool can also output a log of exactly what it is deriving at every stage of the verification process, which we have found to be tremendously helpful when debugging solvers.

Constraint Programming

In constraint programming, we have a more general problem to solve than in the pseudo-Boolean setting. We still have a set of variables, but now variables may take their values from a finite set, rather than being Boolean; we will use capital letters for constraint programming variables, to distinguish them from PB variables. We also have a set of constraints, but these may be in a variety of forms. This generality is a particular strength of constraint programming: in a single model, we may mix Boolean constraints, arithmetic constraints, and other "global" constraints such as "all different". The all different constraint operates on a set of variables of any size, and states that each variable in this set must be given a different value. A given problem could have a single all different constraint, which could operate over some or all of its variables, or it could have many all different constraints, each operating over a different (and potentially overlapping) subset of values. The all different constraint was one of the first global constraints to have a dedicated propagation algorithm (Régin 1994), and remains one of the core constraints present in any constraint programming toolkit—it therefore presents a good minimum standard that any proof logging system must be able to meet.

Compiling Constraint Programming

One approach to solving a constraint programming problem is to compile it to another format, such as Boolean or pseudo-Boolean satisfiability. A simple way of doing so is as follows: for each constraint programming variable X with domain $D(X)$, we create $|D(X)|$ Boolean variables. We then need constraints saying that at least one of these Boolean variables is set to true—this is a disjunction, which may be expressed directly in CNF, or as a single sum inequality in pseudo-Boolean notation. Then we need to mandate that at most one of these Boolean variables is set to true—this is also a sum inequality in pseudo-Boolean notation, but requires all-pairs binary constraints in CNF.

For example, given a constraint satisfaction problem with variables $W \in \{1, 2, 3\}$, $X \in \{2, 3\}$, $Y \in \{1, 3\}$, and $Z \in \{2, 4\}$, we might compile this into OPB format as follows (we omit the header line):

```
* variable W in { 1 2 3 }
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
-1 xW_1 -1 xW_2 -1 xW_3 >= -1 ;
* variable X in { 2 3 }
1 xX_2 1 xX_3 >= 1 ;
-1 xX_2 -1 xX_3 >= -1 ;
* variable Y in { 1 3 }
1 xY_1 1 xY_3 >= 1 ;
-1 xY_1 -1 xY_3 >= -1 ;
* variable Z in { 2 4 }
1 xZ_2 1 xZ_4 >= 1 ;
-1 xZ_2 -1 xZ_4 >= -1 ;
```

Note that for compatibility with pseudo-Boolean solvers, it would be better to use variable names "x1" through "x9"; our tools can also generate numbered variable names, but here will will use more descriptive variable names.

To compile a constraint programming not-equals constraint $X \neq Y$ into either CNF or pseudo-Boolean form, we post a "not both true" constraint for each value that appears in the intersection of the two domains. For example, we could encode $W \neq X$ in the above model using two constraints:

```
* W not equals X, value 2
-1 xW_2 -1 xX_2 >= -1 ;
* W not equals X, value 3
-1 xW_3 -1 xX_3 >= -1 ;
```

Note that no constraint appears for the value 1, which is only present in W's domain.

This suggests a very simple way of compiling an all different constraint: for each distinct pair of variables X and Y in the constraint's scope, we follow the steps to compile a $X \neq Y$ constraint. However, a much more compact encoding is possible in pseudo-Boolean form. For each value that appears in at least one domain, we post a constraint summing over every Boolean variable that corresponds to a CP variable in that constraint taking that value, saying that this sum is at most one. In other words, we are saying that each value can be used at most one time. For example, we could compile an all-different constraint over all four variables as:

```
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
-1 xZ_4 >= -1 ;
```

(The final line could be deleted, because only Z can take the value 4, but leaving it in place reduces the number of special cases needed when implementing a solver.)

Other more sophisticated compilation methods exist, such as those described by Ohrimenko and Stuckey (2008) and Bessiere et al. (2009a). However, these methods are aimed at getting better performance out of solvers, whilst we need only a correct encoding for proof-logging purposes.

Propagators

Constraint programming solvers rarely use these decomposition methods. Instead, solvers have special algorithms called *propagators* associated with each constraint. A propagator can do two things (Schulte and Tack 2009):

1. It can signal that no solution is possible for its associated constraint, based upon the values remaining in the domains of the associated variables.
2. It can remove values from the domains of its associated variables.

[2] https://github.com/StephanGocht/VeriPB/, https://doi.org/10.5281/zenodo.3548582

A propagator may only remove a value from a domain if that value cannot occur in any solution to that constraint. A propagator which will always remove all such values is known as "achieving generalised arc consistency (GAC)" (or sometimes "domain consistency"). For some constraints, achieving GAC is either intractable or impractical, but for the all different constraint GAC may be achieved efficiently and practically (Régin 1994; Gent, Miguel, and Nightingale 2008). Furthermore, GAC for the all different constraint *cannot* be achieved by any polynomial-sized decomposition into Boolean satisfiability (Bessiere et al. 2009b). This is important in practice: there are many examples where strong propagation of constraints is the key to solving hard problems (e.g. Stergiou and Walsh 1999).

From Propagating to Justifying All-Different

The canonical GAC propagation algorithm was introduced by Régin (1994), and has seen considerable subsequent work on how to implement it as efficiently as possible (Gent, Miguel, and Nightingale 2008). We will briefly describe, without proofs, the basic (non-incremental) form of the algorithm, although everything we describe can also be applied to more modern highly tuned implementations. The algorithm works in two stages: firstly, it determines whether it is possible to satisfy the constraint at all, and then if it is, it finds the complete set of values which may safely be deleted from its variables.

Matchings and Hall violators. Let $\{X_1, \ldots, X_N\}$ be the set of variables in an all-different constraint. The *value graph* for this constraint is a bipartite graph, with a vertex in its left set for each variable X_n, and a vertex in its right set for each value that is present in at least one X_n's domain; there is an edge between a variable's vertex and a value's vertex if and only if that variable's domain contains that value. A *matching* is a set of edges in a bipartite graph such that no vertex appears as an endpoint of more than one edge; a matching is *left-saturating* if it covers every vertex on the left, and is of *maximum cardinality* if it contains as many edges as possible.

It is easy to see that left-saturating matchings in a value graph are in one-to-one correspondence with solutions to the all-different constraint. In particular, the constraint can be satisfied if and only if a maximum cardinality matching is left-saturating. Since finding a maximum cardinality matching may be done in polynomial time (Hopcroft and Karp 1973), it is easy to implement a propagator which checks whether or not the constraint is satisfiable.

We are now left with the problem of justifying a backtrack if we find that a maximum cardinality matching is not left-saturating. Using only resolution, this would require exponentially many steps (Haken 1985), but with pseudo-Boolean proofs we are in a better situation. We use Hall's (1935) marriage theorem, which states that a left-saturating matching exists in a bipartite graph if and only if for every subset $W \subseteq \{X_1, \ldots, X_N\}$ we have that $|W| \leq |N(W)|$, where $N(W)$ denotes the neighbourhood of W. In particular, if a left-saturating matching does not exist, then there exists a *Hall violator* W where $|N(W)| < |W|$; in our terms, this is a set of n variables whose domains contain strictly fewer than n values between them.

A conventional propagator does not care about the existence of Hall violators, and only looks at the size of a maximum cardinality matching. However, the usual augmenting paths algorithm for finding a maximum cardinality matching can easily be extended to output a Hall violator by following an alternating path backwards from an unmatched left-vertex.

Given such a set of variables H, a justifying propagator must be able to express that "either one of the variables in H must be given a value that is currently not present in its domain, or there is a contradiction". To do this, we count sets of variable-value pairs in two different ways. Firstly, we have (from the model) that each variable in H must be given at least one value—call these constraints $AL1(h)$. We sum together these constraints, to achieve an expression of the form $\sum_{h \in H} AL1(h) \geq |H|$. Now, letting $D(H)$ mean the values in the union of the domains of the variables in H, and denoting the "value can be used at most once" constraints from the model as $AM1(v)$, we sum these to get $\sum_{v \in D(H)} AM1(v) \leq |D(H)|$. Since H is a Hall violator, $|H| > |D(H)|$, so the sum of these two sums gives a suitable justification.

Continuing our running example, suppose that the Z variable could not take the value 4, due to it being eliminated by another constraint or by a guessed assignment during search. In this case, a maximum cardinality matching in the value graph would leave a single variable uncovered. Suppose the matching found is $\{W = 1, X = 2, Y = 3\}$ leaving Z uncovered. In this case, the Hall violator has the four variables $\{W, X, Y, Z\}$, and the three associated values are $\{1, 2, 3\}$. By summing up the lines saying

```
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
1 xX_2 1 xX_3 >= 1 ;
1 xY_1 1 xY_3 >= 1 ;
1 xZ_2 1 xZ_4 >= 1 ;
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
```

using a proof logging command which could look like (if the lines for the variable axioms for W, X, Y and Z are 1, 3, 5 and 7, and the all-different constraint starts on line 9):

```
p 1 3 + 5 + 7 + 9 + 10 + 11 + 0
```

we derive the constraint

```
1 xZ_4 >= 1 ;
```

which means that Z must take the value 4 after all—and if it cannot then we have proved unsatisfiability.

Strongly connected components and Hall sets. The second stage of the propagation process takes place only if a left-saturating matching has been found. If such a matching M exists, a new *directed* bipartite graph known as the *residual graph* is created by taking the value graph, and direct-

ing edges as right-to-left if they are present in M and left-to-right otherwise. This graph has the property that certain edges that start in one strongly connected component and end in another correspond to variable-value assignments that will never appear in any maximum cardinality matching—we refer to Gent, Miguel, and Nightingale (2008) for full details.

Again, we cannot directly express graph-theoretic properties in a proof log, but a connection between combinatorics and graph theory saves us. Every edge which describes a deletion is due to the existence of a Hall set—that is, a set of n variables whose union contains exactly n values (Quimper and Walsh 2005). More specifically, there is no solution to an all-different constraint where variable X_i gets value v_j if and only if there exists a set H of variables not including X_i whose domains contain exactly $|H|$ values between them, one of which is v_j.

Given a Hall set H, we may output a pseudo-Boolean constraint justifying the deletions it triggers by following the same process as for a Hall violator: we sum up the "variable must be given at least one value" constraints and the "value must be used at most once" constraints, and this time arrive at an equality which shows that no variable outside of the Hall set may be given any value in the Hall set.

It remains only to identify the relevant Hall sets, which is also straightforward: they correspond precisely to the strongly connected components in the residual graph which have a deletion edge entering them (Dulmage and Mendelsohn 1958). Note that a single Hall set can justify multiple deletions (and for space reasons it is advantageous to detect this and avoid emitting duplicate constraints).

Returning to our running example, the variables $\{W, X, Y\}$ form a Hall set with three values $\{1, 2, 3\}$. The Z variable also includes the value 2, which may be deleted. We may justify this by summing the lines

```
1 xW_1 1 xW_2 1 xW_3 >= 1 ;
1 xX_2 1 xX_3 >= 1 ;
1 xY_1 1 xY_3 >= 1 ;
-1 xW_1 -1 xY_1 >= -1 ;
-1 xW_2 -1 xX_2 -1 xZ_2 >= -1 ;
-1 xW_3 -1 xX_3 -1 xY_3 >= -1 ;
```

using a command like

```
p 1 3 + 5 + 9 + 10 + 11 + 0
```

to derive a new constraint

```
1 ~xZ_2 >= 1 ;
```

which corresponds to saying that Z may not take the value 2. If we had another variable Q with domain $\{2, 5, 6\}$, the constraint generated using the same sum would instead be

```
1 ~xZ_2 1 ~xQ_2 >= 2
```

showing that neither Z nor Q could take the value 2.

Finally, we note that Hall sets may nest. For example, given $A, B \in \{1, 2\}$, $C, D \in \{1, 2, 3, 4\}$, and $E \in \{1, 2, 3, 4, 5\}$, the process we describe would output Hall sets $\{A, B\}$ and $\{C, D\}$, *not necessarily in that order*. This does not matter for our purposes (so long as we are using the "u" proof rule rather than a series of "p" steps to describe the search tree, as discussed in the following section). However, if it would for some reason be preferable to output Hall sets $\{A, B\}$ and $\{A, B, C, D\}$ (which justify the same deletions from E, independently of the order in which they are carried out), this may be done by outputting every vertex in the residual graph which is reachable from the end of the deletion edge, rather than looking at strongly connected components.

Justifications Versus Explanations

Much of what we have discussed resembles the *explanations* produced by lazy clause generation constraint programming solvers (Ohrimenko, Stuckey, and Codish 2009; Downing, Feydy, and Stuckey 2012). Lazy clause generation solvers will create new clauses on the fly as the result of propagations, allowing for SAT-style conflict analysis to be mixed with constraint programming propagation. However, explanations can be created "out of nowhere" without justification: an explaining propagator merely asserts that the clause it produces is valid, and does not have to demonstrate its derivation. Nonetheless, there is potential for crossover between these two areas going forward: in one direction, perhaps generating more expressive PB constraints and using PB conflict analysis will lead to better lazy clause generation solvers, and in the other direction, it may be possible to reduce the amount of work needed to produce justifying propagators by building upon what is known about explanations.

Another related piece of work is the constraint programming solver described by Veksler and Strichman (2010), which fits somewhere in between justifications and explanations. This solver produces proof logs, but in a format which requires the proof verifier to support specialised inference rules for every new global constraint. In contrast, our approach shows that practical proof logging is still possible even without requiring the proof verifier to know about the propagation behaviour of any global constraint.

A Justifying Constraint Programming Solver

Finally, we briefly describe the implementation of a small constraint programming solver which can output a justification of all of the choices it makes.[3] This solver is implemented in C++, and supports the all different constraint with full GAC propagation, as well as equals, not-equals, and (forward checking) table constraints. The solver has not been designed for performance, but rather to identify the best engineering decisions for implementing a proof logging solver.

The solver differs from a conventional constraint programming solver in three areas: being able to compile models to the pseudo-Boolean format, being able to log search operations, and being able to log propagation.

Compilation. As well as solving a constraint programming model, a proof-logging solver must be able to trans-

[3] https://github.com/ciaranm/certified-constraint-solver, https://doi.org/10.5281/zenodo.3549712

late the model into an equivalent pseudo-Boolean model. We described the theory behind this in the previous section. From an implementation perspective, this was reasonably straightforward: the solver must track the PB variable naming used for each variable-value which it encodes, and some constraints must remember the line number used when outputting some of their rules (for example, the all different constraint must be able to recall the appropriate "at most one" line for each of the values in its scope).

Search. Proof logging during search is remarkably simple when using reverse unit propagation. Whenever the solver backtracks (either due to propagation failure, or a domain wipeout), it suffices to output a proof line of the form

```
u -1 xA_3 -1 xB_4 -1 xC_1 >= -2 ;
```

where the variables are the decision variables on the solver's trail—that is, the solver is asserting that whatever it just guessed "obviously" leads to a contradiction, and so at least one of the guessed assignments must be incorrect. Ultimately, this leads to the solver outputting

```
u >= 1 ;
```

after backtracking on the first decision variable, which can then be followed by the assertion of contradiction.

Propagation. Due to reverse unit propagation, it is not necessary to make any changes at all for the equals and not-equals constraints—these propagations need not be logged. For the all different constraint, it suffices to output "p" rules for every Hall violator and Hall set which leads to a contradiction or propagation, as described in the previous section. (Again due to the use of reverse unit propagation, there is no need to adapt these constraints to mention the trail.)

We believe this demonstrates the simplicity of proof logging in this format. In earlier prototypes not making use of reverse unit propagation, the burden upon the solver writer was *vastly* greater, with a trail-aware "p" rule being required for every single propagating step.

Experiments

To test our solver and proof verifier, we generated a number of 25×25 unsatisfiable Sudoku instances. Solving such instances in a reasonable amount of time requires the full capabilities of all-different propagation, and tests all of the functionality of our tools. For a representative instance which required 1,691 guessed decisions to solve, our solver took 41 seconds to prove unsatisfiability, which increased to 42 seconds when logging a proof (we stress that this implementation has not been designed for performance). The proof log contained 109,519 Hall set propagations, and 846 Hall violators, and could be verified in 6 seconds.

We also tried deliberately introducing bugs into our solver—for example, by failing to find maximum cardinality matchings that required two augmenting steps, and by randomly omitting logging for a small number of Hall sets. In each case the proof verifier caught the mistakes, although only if the instances selected actually triggered the faulty behaviour. (For example, it is surprisingly rare for multiple augmenting path steps to be required to find a maximum cardinality matching, when starting from a greedy matching.) Because our solver was designed from the ground up with proof logging, we were also able to use proof logs to catch bugs early on in the development process that had not been detected by conventional testing techniques.

Conclusion

We have shown that it is both possible and practical for a constraint programming solver to produce a pseudo-Boolean proof log for unsatisfiability, even when all different constraints are in use. This is unexpected: pseudo-Boolean reasoning knows nothing about graphs, matchings, augmenting paths, or strongly connected components, all of which are required for all different propagation. This suggests that we should be more broadly interested not just in algorithms for propagation, but in languages for justifying propagation—unlike in the Boolean satisfiability community, these concepts are not equivalent. We therefore intend to investigate which other families of global constraint can be justified easily using pseudo-Boolean reasoning. Obviously, any constraint for which we already know a strongly-propagating SAT or pseudo-Boolean encoding requires no further work, but we believe that several other common constraints are also justifiable.

One might ask whether a new proof format is really necessary. The main difference to the existing DRAT proof format used by SAT solvers is that we are using cutting planes proofs instead of resolution proofs. This makes it very simple to express the counting arguments we are using to justify propagations and conflict of the all-different constraint. This kind of reasoning cannot be done efficiently with resolution. However, DRAT allows the introduction of new variables, which is known to be very powerful. (Our proof format currently does not have this capability but an extension is in progress.) In theory, using additional variables allows DRAT to verify cutting planes reasoning and hence to justify the all-different constraint.

A natural and interesting question, therefore, is how DRAT proof logging would compare to our approach. As we already mentioned previously, though, the problem is that DRAT proof logging for cardinality reasoning is a *theoretical* result. The fact that DRAT logging can be done in principle, with at most a polynomial blow-up, does not mean that it is possible to do in practice, and to the best of our knowledge no-one has been able to produce any implementation that can be used to run practical experiments. This means that we cannot compare the performance of our pseudo-Boolean proof logging with DRAT proof logging, not because DRAT would run so much more slowly, but because it is so much more complicated that no-one has even implemented it. In contrast, our pseudo-Boolean proof logging is both fast and simple.

The approach we describe does still require the user to trust that the pseudo-Boolean model file produced corresponds exactly to the high level constraint programming model given as input. This should not necessarily be taken as given—not all global constraints can be encoded in as

straightforward a manner as all different, and additionally the compilers for higher-level constraint modelling languages such as Essence and MiniZinc could introduce further bugs. It may therefore be worthwhile to investigate techniques from conventional compilers to verify this part of the process.

We stress that our approach does not prove that a solver is correct—it simply ensures that if a solver ever produces an incorrect answer, then this can be detected and a human brought in to fix the problem. On the other hand, when a justifying solver does produce a correct answer by legitimate means, the proof can be archived for posterity. We can thus always be confident that the answer is indeed correct, even if we do not trust the solver that produces the proof or the person who is claiming that the proof was produced by a trustworthy solver. And finally, we note that proof logging will catch more esoteric problems such as compiler bugs, hardware errors, and cosmic rays that could make a correct solver output an incorrect answer.

Acknowledgments

The first and second authors were funded by the Swedish Research Council (VR) grant 2016-00782. The fourth author was also supported by the Knut and Alice Wallenberg grant KAW 2016.0066 and the VR grant 621-2012-5645. The third author was supported by the Engineering and Physical Sciences Research Council [grant number EP/P026842/1].

References

Akgün, Ö.; Gent, I. P.; Jefferson, C.; Miguel, I.; and Nightingale, P. 2018. Metamorphic testing of constraint solvers. In Hooker, J. N., ed., *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, 727–736. Springer.

Bessiere, C.; Katsirelos, G.; Narodytska, N.; Quimper, C.; and Walsh, T. 2009a. Decompositions of all different, global cardinality and related constraints. In Boutilier (2009), 419–424.

Bessiere, C.; Katsirelos, G.; Narodytska, N.; and Walsh, T. 2009b. Circuit complexity and decompositions of global constraints. In Boutilier (2009), 412–418.

Blake, A. 1937. *Canonical Expressions in Boolean Algebra*. Ph.D. Dissertation, University of Chicago.

Boutilier, C., ed. 2009. *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*.

Buss, S., and Nordström, J. 2019. Proof complexity and SAT solving. Chapter to appear in the 2nd edition of *Handbook of Satisfiability*, Draft version available at https://www.math.ucsd.edu/~sbuss/ResearchWeb/ProofComplexitySAT/.

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

Davis, M., and Putnam, H. 1960. A computing procedure for quantification theory. *J. ACM* 7(3):201–215.

Downing, N.; Feydy, T.; and Stuckey, P. J. 2012. Explaining alldifferent. In Reynolds, M., and Thomas, B. H., eds., *Thirty-Fifth Australasian Computer Science Conference, ACSC 2012, Melbourne, Australia, January 2012*, volume 122 of *CRPIT*, 115–124. Australian Computer Society.

Dulmage, A. L., and Mendelsohn, N. S. 1958. Coverings of bipartite graphs. *Canadian Journal of Mathematics* 10:517–534.

Gent, I. P.; Miguel, I.; and Nightingale, P. 2008. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.* 172(18):1973–2000.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. Solvercheck: Declarative testing of constraints. In *CP 2019, Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming*. To appear.

Goldberg, E. I., and Novikov, Y. 2003. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference (DATE)*, 10886–10891. IEEE Computer Society.

Haken, A. 1985. The intractability of resolution. *Theor. Comput. Sci.* 39:297–308.

Hall, P. 1935. On representatives of subsets. *Journal of the London Mathematical Society* s1-10(1):26–30.

Heule, M.; Hunt Jr., W. A.; and Wetzler, N. 2013a. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 181–188. IEEE.

Heule, M.; Hunt Jr., W. A.; and Wetzler, N. 2013b. Verifying refutations with extended resolution. In Bonacina, M. P., ed., *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, 345–359. Springer.

Heule, M.; Järvisalo, M.; and Suda, M. 2019. The international SAT Competitions web page. http://www.satcompetition.org.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Creignou, N., and Berre, D. L., eds., *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, 228–245. Springer.

Hopcroft, J. E., and Karp, R. M. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2(4):225–231.

Lamb, E. 2016. Two-hundred-terabyte maths proof is largest ever. *Nature* 545:17–18.

Marques Silva, J. P., and Sakallah, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* 48(5):506–521.

Ohrimenko, O., and Stuckey, P. J. 2008. Modelling for lazy clause generation. In Harland, J., and Manyem, P., eds., *Theory of Computing 2008. Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia, January 22-25, 2008. Proceedings*, volume 77 of *CRPIT*, 27–37. Australian Computer Society.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14(3):357–391.

Quimper, C., and Walsh, T. 2005. The all different and global cardinality constraints on set, multiset and tuple variables. In Hnich, B.; Carlsson, M.; Fages, F.; and Rossi, F., eds., *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and*

Invited Papers, volume 3978 of *Lecture Notes in Computer Science*, 1–13. Springer.

Régin, J. 1994. A filtering algorithm for constraints of difference in CSPs. In Hayes-Roth, B., and Korf, R. E., eds., *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, 362–367. AAAI Press / The MIT Press.

Schulte, C., and Tack, G. 2009. Weakly monotonic propagators. In Gent, I. P., ed., *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, 723–730. Springer.

Stergiou, K., and Walsh, T. 1999. The difference all-difference makes. In Dean, T., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, 414–419. Morgan Kaufmann.

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

Veksler, M., and Strichman, O. 2010. A proof-producing CSP solver. In Fox, M., and Poole, D., eds., *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

Wetzler, N.; Heule, M.; and Hunt Jr., W. A. 2014. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Sinz, C., and Egly, U., eds., *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, 422–429. Springer.

Paper F

An Auditable Constraint Programming Solver

Stephan Gocht
Lund University, Sweden
University of Copenhagen, Denmark

Ciaran McCreesh
University of Glasgow, Scotland

Jakob Nordström
University of Copenhagen, Denmark
Lund University, Sweden

Abstract

We describe the design and implementation of a new constraint programming solver that can produce an auditable record of what problem was solved and how the solution was reached. As well as a solution, this solver provides an independently verifiable proof log demonstrating that the solution is correct. This proof log uses the VeriPB proof system, which is based upon cutting planes reasoning with extension variables. We explain how this system can support global constraints, variables with large domains, and reformulation, despite not natively understanding any of these concepts.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Discrete optimization

Keywords and phrases Constraint programming, proof logging, auditable solving

Supplementary Material Source code for the solver described in this paper can be found here:
Software: `https://doi.org/10.5281/zenodo.6514093`

Funding *Stephan Gocht*: Supported by the Swedish Research Council grant 2016-00782
Ciaran McCreesh: Supported by a Royal Academy of Engineering research fellowship
Jakob Nordström: Supported by the Swedish Research Council grant 2016-00782 and the Independent Research Fund Denmark grant 9040-00389B

1 Why Trust a Constraint Programming Solver?

Proof logging is now a standard practice in the Boolean satisfiability (SAT) community: alongside a solution, solvers are expected to produce a proof, in a standard format called DRAT [19, 18, 33], that can be verified independently to ensure that the correct answer was reached through legitimate reasoning. As well as reducing the number of bugs in solvers, this has been vital for the social acceptability of computer-generated mathematical proofs [21]. These successes mean that proof logging is now being considered in other areas, including mixed integer programming [9] and subgraph-finding algorithms [14, 13], and a similar paradigm known as *certifying algorithms* exists for polynomial-time solvable problems [23].

We believe that a practical proof logging system would also be extremely useful for the constraint programming (CP) community. In the 2021 MiniZinc challenge, at least 45 out of 3,500 claimed solutions were incorrect (either through falsely claiming unsatisfiability or optimality, or by providing infeasible "solutions"), and previous years saw similar rates. Furthermore, this was not limited to one solver, one problem, or one global constraint. Although this high error rate does not necessarily reflect what we might see in practice, it strongly suggests that we should not be complacent. And even if we are completely convinced that our solvers are correct, thanks to extensive testing using domain-specific methods [1, 12], there are still benefits to be had from proof logging. When CP is used for life-affecting decision-making, having a solver that can produce an independently verifiable record of

what the problem was and how it was solved would be much better for public confidence in algorithms than saying "trust us, we tested it carefully". In effect, we would be making the solving process *auditable*, and removing the need for trust.

In some applications, compiling a CP model to SAT and re-using SAT proof logging might be a viable approach for auditability. However, this is not a universal solution: even if the loss of solving power from switching representations is not a problem, why should we trust that a complex compilation process is correct? And what if we need to solve enumeration or optimisation problems, neither of which are supported by DRAT? Nor is it practical to make CP solvers output DRAT proofs, even for decision problems: attempts at expressing the strong reasoning carried out by simple global constraints like all-different have introduced intolerable overheads [28, 10], and DRAT does not seem well-suited even for the parity reasoning done by some modern SAT solvers [15]. One alternative would be to introduce a much stronger and more complex proof format, that is aware of the meaning of every global constraint and every kind of propagation that could be performed [31] and every kind of reformulation ever invented—but why should we trust such a complicated proof logging system, and how would we even know that it is consistent? This is not a trivial concern: even the relatively simple DRAT format has had issues in this respect [26].

This paper describes an alternative approach to proof logging which addresses all of these problems. It uses an existing proof verifier, VeriPB, which was designed for pseudo-Boolean models. VeriPB's proof format uses cutting planes reasoning [5] and redundance-based strengthening [15], which is only a small step up in complexity from the DRAT approach of using Boolean models and extended resolution. However, this small change in underlying proof system suddenly means that proof logging for many kinds of constraint becomes both efficient and easy to implement, despite the system not having any explicit notion of global constraints or even non-binary variables. Thanks to this, we have been able to implement a new prototype constraint programming solver which can produce proof logs for all of its reasoning, with support for global constraints like all-different, integer linear inequality (including for variables with very large domains), table, minimum / maximum of an array, element, and absolute value, as well as some simple automatic reformulation.

Our aim in this work is not to produce the world's fastest solver, but rather to explore the design decisions necessary to provide auditable solving when operating with diverse constraints, and to explain how to understand and adopt proof logging technology. The main differences between our solver and a basic conventional solver such as MiniCP [24] are:

- The solver can describe the semantics of variables and each constraint in a low-level format, which is discussed in Section 2.1. We give examples in Section 3. This is the only part of the process that is not directly auditable: we discuss this further in Section 5.1.
- Whenever the solver backtracks during search, it creates a proof step asserting that its current sequence of guesses "obviously" (but verifiably) implies a contradiction, as described in Section 2.3. When enumerating or optimising, solutions must also be logged.
- Any piece of code that potentially removes a value from a variable's domain (or instantiates it, or changes its bounds) must be able to provide a justification that can be added to the proof log. This justification can be "this is immediately obvious", "use reverse unit propagation" (Section 2.3), or occasionally, "use the following explicit proof steps" (Section 2.2). In many ways this resembles lazy clause generation solvers [25], except that justifications must be derived in a sound and verifiable manner, rather than being introduced from nowhere. We give examples in Section 4.
- Finally, some constraints make use of reformulations, which must also be justified; examples are in Section 4.5.

Together, these additions mean that if the solver ever produces an incorrect answer, this can be detected—even if it is due to a compiler or hardware fault rather than a solver bug. Our results demonstrate that proof logging for constraint programming is rapidly becoming a technologically viable approach, and one that will be worth adopting in other solvers. We conclude by outlining how we might realise this goal of auditable constraint solving.

2 The VeriPB Proof System

We begin with an overview of the relevant parts of the model and proof system used by VeriPB.[1] It is important to stress immediately that solvers do not need to understand or implement this proof system (in the same way that most SAT solvers do not "know" that they are searching for resolution proofs). Indeed, the prototype solver we describe later in this paper produces proofs through templates, never manipulates proof steps directly, and does not know enough about the VeriPB proof system to be able to verify its own proofs.

In this section we will primarily be talking about proofs of unsatisfiability. Both VeriPB and our solver also support optimisation, enumeration, and satisfiable decision problems, but the core of the proof system concerns unsatisfiability. The idea behind a proof is that we start off with known facts, which come from the input model. Then, at each step in the proof, we derive a new fact which is "obviously" a consequence of some combination of previous facts. We finish by deriving a fact which is clearly a contradiction, which in turn means it must be the case that the input is unsatisfiable.

2.1 Pseudo-Boolean Models

VeriPB takes as input a pseudo-Boolean model, which is a very restricted kind of constraint programming model. A pseudo-Boolean model is defined by a set $\{x_i\}$ of $\{0,1\}$ integer variables, and a set of integer linear inequalities $\sum_i c_i x_i \geq n$ for integers c_i and n. In this paper we will use lowercase variable names to refer to pseudo-Boolean variables, and uppercase variable names to refer to constraint programming variables. We will also write some constraints using $\leq$ instead of $\geq$, and will write $\sum_i c_i x_i = n$ as shorthand for two inequalities. We use the convention that $x = 0$ means false, and $x = 1$ means true; we write $\overline{x}$ to mean $1 - x$. Observe that Boolean satisfiability constraints in conjunctive normal form (CNF) can easily be written as pseudo-Boolean constraints, because e.g. $(x_1 \vee \overline{x}_2 \vee x_3)$ holds if and only if $(x_1 + \overline{x}_2 + x_3 \geq 1)$. For clarity we will sometimes mix logical and pseudo-Boolean notation, and write expressions like $(x_1 \wedge x_2 \wedge x_3) \rightarrow (2x_4 + 3x_5 + -4x_6 \geq 7)$ rather than the more cumbersome $1\overline{x}_1 + 1\overline{x}_2 + 1\overline{x}_3 + 2x_4 + 3x_5 + -4x_6 \geq 7$.

There is a standard textual file format for pseudo-Boolean models, known as OPB [27]. VeriPB supports this format, with extensions: for example, it allows variables to have descriptive names, which is convenient for readability, and can include implications to avoid the need for solver authors to calculate appropriate coefficients manually.

2.2 Cutting Planes

Alongside an OPB file, VeriPB takes a proof log file that claims to show that the pseudo-Boolean model is unsatisfiable, and checks the proof's validity. This proof log is a text file, which describes a sequence of steps using the cutting planes proof system [5]. In cutting

[1] `https://gitlab.com/MIAOresearch/VeriPB`

planes, we can add two constraints together, multiply a constraint by a non-negative integer constant, and divide existing constraints by a positive integer constant (with rounding); we may also assert that any literal is non-negative. The aim is to derive a constraint saying that $0 \geq 1$, which serves as a contradiction. The cutting planes proof system is complete for pseudo-Boolean models, in the same way that resolution is complete for Boolean models. However, it is exponentially stronger than resolution: for example, resolution requires exponential length proofs for all-different constraints, whereas cutting planes can justify Hall set reasoning in (small) polynomial length [10]. For more details on the theoretical background, see, e.g., the survey by Buss and Nordström [3].

2.3 Unit Propagation and Reverse Unit Propagation

For solver authors, working directly with cutting planes can be difficult, and would require every part of a solver to keep careful track of every operation carried out. This difficulty can be avoided through the use of *reverse unit propagation* (RUP) proof steps [16, 30, 10], which are in effect shorthand for a sequence of cutting planes steps.

For CNF clauses, *unit propagation* means identifying any clause where all but one of its literals has already been set the wrong way, and propagating the remaining literal to the value that avoids violating the clause, repeating until either a contradiction is reached or no further unit clauses exist. This notion generalises to pseudo-Boolean constraints, where unit propagation means achieving integer bounds consistency [4]. A constraint C is said to be RUP if asserting its negation leads to a contradiction via unit propagation; in such a case, it is obviously permissible to introduce C as a new constraint without altering whether the underlying model is satisfiable.

RUP steps in a Boolean setting form the core part of the DRAT proof format. This is useful for solver authors because for a typical CDCL SAT solver, every learned clause is RUP, and so writing a proof log requires only that a solver output every clause it learns in turn. In our constraint programming setting, RUP clauses will similarly form the backbone of the proofs we generate, with a RUP clause being written every time a solver backtracks. However, we will also use explicit cutting planes steps where necessary, to justify complex propagations. In one sense, RUP is purely a convenience for solver authors, in that with more work, cutting planes steps could be used instead; however, this would require substantially more book-keeping in the solver.

The following pieces of intuition may be helpful in what follows: a fact follows from unit propagation if it is so immediately obvious that it is not worth stating. A fact follows from reverse unit propagation if, once you have been told that it is a fact, it is obviously true (but that it might not be immediately obvious if you are not told). In some ways this resembles failed literal probing, or the difference between generalised arc consistency and singleton arc consistency; this intuition may become clearer following the example in Section 4.1.

2.4 Extension Variables and Redundance-Based Strengthening

An *extension variable* is a new variable introduced as part of a proof. In VeriPB, extension variables are supported using a rule called *redundance-based strengthening* (which, for readers familiar with SAT proof logging, is the natural analogue of the RAT rule in DRAT) [15]. We do not need the full power or definition of that rule for this paper. It suffices to say that, for an arbitrary pseudo-Boolean constraint C and a new variable y that has not previously appeared in the model or proof, we are allowed to introduce the reified constraints $y \leftrightarrow C$

at any point during the proof. As well as being extremely convenient for solver authors, extension variables also give an exponential increase in reasoning power [3].

2.5 Satisfiable Instances, Enumeration, Optimisation, and Deletions

For satisfiable decision problems, VeriPB supports solution checking by including a solution in a proof log. Enumeration problems may also be verified this way: whenever a solution is logged, VeriPB treats this as introducing a new constraint saying "but not this solution", and so a proof is effectively a proof by contradiction that there are no solutions other than the ones listed. Optimisation is handled similarly, via an optional objective expression in the OPB file. Finally, in practice it is important to delete constraints from the proof that will not be re-used later on. This is straightforward, but will not be discussed in this paper.

3 Encoding Constraint Programming Models

In the previous section, we learned that if we wish to use VeriPB to verify constraint programming proofs then we must provide two things: a pseudo-Boolean model in OPB format, and a proof log. We now discuss how the first of these two steps may be generated by a CP solver, looking first at how we turn CP variables into pseudo-Boolean variables, and then at how we represent constraints. When compiling CP to a lower level format for solving, selecting a good encoding involves considering propagation and consistency; in contrast, for proof logging we need only something that is simple and reasonably compact.

3.1 Variables

The most straightforward way of encoding an integer variable X with domain $\ell \dots u$ is to create $u - \ell + 1$ pseudo-Boolean variables $x_{=i}$, where $x_{=i}$ is true if and only if $X = i$, together with supporting constraints saying that $\sum x_{=i} = 1$. Such an approach was used for proof logging the all-different constraint by Elffers et al. [10]. However, this is impractical for variables with large domains that are only involved in bounds-consistent constraints such as integer linear inequalities. Instead, we define a binary encoding. Let h be the least strictly positive number such that $2^{h-1} \geq \max(1, |u|, |\ell|)$. Then we introduce variables $x_{\mathrm{b}i}$ for $i \in 0 \dots h-1$, and, if $\ell < 0$, we additionally introduce an x_{neg} variable to give us a two's complement style representation. The two constraints $\ell \leq -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} 2^i x_{\mathrm{b}i} \leq u$ then define the meaning and bounds of these variables (with the leading sum term omitted if $\ell \geq 0$). For example, if we have a constraint programming variable A with domain $\{-3 \dots 9\}$, we would define

$$-32a_{\mathrm{neg}} + 1a_{\mathrm{b}0} + 2a_{\mathrm{b}1} + 4a_{\mathrm{b}2} + 8a_{\mathrm{b}3} + 16a_{\mathrm{b}4} \geq -3 \text{ and}$$
$$32a_{\mathrm{neg}} + -1a_{\mathrm{b}0} + -2a_{\mathrm{b}1} + -4a_{\mathrm{b}2} + -8a_{\mathrm{b}3} + -16a_{\mathrm{b}4} \geq -9.$$

Although compact, experienced modellers know that such an encoding often leads to extremely poor propagation. This is a problem if the encoding is to be used for solving, but for proof logging this is not an issue because the encoding only restricts how we write a proof, not how a solution is reached. However, when expressing constraints or propagation, it is often useful to be able to use variables $x_{=i}$ and $x_{\geq i}$ for selected values of i. If these variables are used when the constraints appear in the pseudo-Boolean model, we can define them immediately. We have found the most convenient way of expressing these variables to be

$$x_{\geq i} \leftrightarrow -2^h x_{\mathrm{neg}} + \sum_{i=0}^{h-1} x_{\mathrm{b}i} \geq i$$

and similarly for $x_{\geq i+1}$. Additionally, we constrain that $x_{\geq i+1} \rightarrow x_{\geq i}$, and force $x_{\geq i}$ to be true or false respectively if i defines a lower or upper bound. We then define $x_{=i} \leftrightarrow x_{\geq i} \wedge \overline{x}_{\geq i+1}$.

However, what if these values are only used for branching or propagation, such as when dealing with linear inequalities (discussed below)? The whole point of using a binary encoding was to avoid having to define variables for values that never appear in a constraint or a proof. Fortunately, it is possible to introduce these additional variables as extension variables with the same defining constraints, so long as it is done in exactly the order described above. In such a case, we also introduce the RUP constraints $x_{\geq i} \rightarrow x_{\geq j}$ and $x_{\geq h} \rightarrow x_{\geq i}$ for the closest two values h and j that already have equality variables, if they exist. Finally, when propagating certain constraints such as all-different, it is also convenient to have an at-least-one constraint $\sum_{i=\ell}^{u} x_{=i} \geq 1$. If all the $x_{=i}$ variables are defined, then this constraint can also be introduced via RUP as needed, and does not need to be defined in the model. This can make the pseudo-Boolean model much more manageable: for example, for the implementation of the "cake" problem discussed in Section 5, our solver introduces a total of one hundred $x_{=i}$ or $x_{\geq i}$ variables in the proof, rather than defining several hundred thousand of them in the OPB file.

Note finally that the details of this encoding are largely irrelevant to most constraints. In particular, it is possible for the part of a solver that deals with proof logging to treat 0/1 variables separately with almost no impact on the rest of its code.

3.2 Constraints

Next, we must represent every constraint in pseudo-Boolean form. This topic is relatively well-understood, because pseudo-Boolean constraints are a superset of CNF—and again, it is not necessary to find a *good* encoding, only a simple and correct one. We now give some examples that illustrate general concepts.

Integer linear inequalities can easily be expressed in pseudo-Boolean form by adding multiples of the bit encodings together. For example, suppose we have the CP constraint $2A + 3B + 4C \leq 42$, where each of the variables has domain $\{-3 \dots 9\}$. This would be translated into

$$\begin{aligned} &-64a_{\text{neg}} + 2a_{\text{b0}} + 4a_{\text{b1}} + 8a_{\text{b2}} + 16a_{\text{b3}} + 32a_{\text{b4}} \\ &+ -96b_{\text{neg}} + 3b_{\text{b0}} + 6b_{\text{b1}} + 12b_{\text{b2}} + 24b_{\text{b3}} + 48b_{\text{b4}} \\ &+ -128c_{\text{neg}} + 4c_{\text{b0}} + 8c_{\text{b1}} + 16c_{\text{b2}} + 32c_{\text{b3}} + 64c_{\text{b4}} >= 42. \end{aligned}$$

We may use a pair of such constraints to define equality and sum constraints. If we were solving using these constraints, we would get very weak propagation, but we will explain why this does not matter in the following section.

Not equals constraints can be expressed using two *half-reified* linear inequalities: we introduce a Boolean flag f, and define the constraints $f \rightarrow (A - B > 0)$ and $\overline{f} \rightarrow (B - A > 0)$. These can be expressed in pseudo-Boolean form as integer linear inequalities with the addition of a suitably large coefficient on the negation of the flag to handle the implication. A similar encoding can be used for the absolute value constraint.

All-different can be expressed by a set of at-most-one constraints, such as $a_{=2} + b_{=2} + c_{=2} \leq 1$, or by a clique of not-equals constraints. Again, solving using either kind of constraint would

give weaker propagation than the usual GAC all-different constraint, but this is not a concern for proof logging.

Table constraints can be expressed in terms of an auxiliary variable, which selects which tuple is matched. For example, given the tuple sequence $[(1,2,3),(1,3,4),(2,2,5)]$ applied as a table constraint to the variables $[A,B,C]$, we could express this by adding an auxiliary variable $T \in \{0\dots2\}$ (called the *tuple selector variable*), and using implication constraints $t_{=0} \rightarrow (a_{=1} \wedge b_{=2} \wedge c_{=3})$ etc.

Element constraints come in a variety of forms. For example, consider a 2D element from constants constraint, which says that a variable V takes the $[I,J]$th entry in a two dimensional $m \times n$ array A of constants. This shows up in various places, such as the MiniCP quadratic assignment problem benchmark discussed in Section 5 (where solvers are expected to achieve generalised arc consistency on the two index variables, and bounds consistency on the assigned variable) [24]. The only constraints we will define are the unary constraints $i_{\geq 0}$, $\overline{i}_{\geq m}$, $j_{\geq 0}$ and $\overline{j}_{\geq n}$, and then $(i_{=x} \wedge j_{=y}) \rightarrow v_{=A[x,y]}$ for each array entry. Such constraints, on their own, obviously do not enforce the desired consistency levels, but they have the advantage of being simple. This technique also generalises. For example, if the array A is not constant then the implication constraints can become half-reified equalities instead—and this in turn makes it easy to define array minimum and array maximum constraints.

Other constraints are usually similarly easy to express. The critical point is that encodings need only be correct, not good, and so if we know how to express the constraint at all in CNF or as integer linear inequalities then that is sufficient. Similarly, if a constraint easily fits in a table, then it can be handled that way. Combined with the ability to use auxiliary variables, even constraints like "forms a connected subgraph" are manageable [13].

4 Proofs for Search and Propagation

The core of a proof for an unsatisfiable constraint satisfaction problem is a description of the solver's search tree. This is expressed as a RUP statement for every backtrack, and ends with a contradiction when we backtrack from the root node. The idea is that whenever the constraint solver backtracks, it should be "obvious" that the sequence of guesses made leads to a dead end, and is thus a RUP clause. Gocht et al. [13] provide a worked example of this process in a branch and bound setting for a clique algorithm.

In order to make this process work with global constraints, we will need to include additional proof statements to justify non-obvious propagations (in the same way that Gocht et al. had to justify the clique algorithm's bounds). The core invariant we use is that at every backtrack, any variable-value deletion that is known to the CP solver (and thus part of the decision to backtrack) must be visible to the proof verifier either through unit propagation, or through reverse unit propagation of the backtrack clause. This section elaborates on what this means for various global constraints.

4.1 RUP Justifications and Table Constraints

Achieving generalised arc consistency for table constraints involves two kinds of inference: detecting when a tuple becomes infeasible, and detecting when a variable's value is no longer supported by any feasible tuple. There are several different propagation algorithms for

performing this inference [29, 22, 7, 20, 32], but from a proof logging perspective it does not matter how the inference is performed, only what is inferred.

A tuple becoming infeasible requires no justification. Recall that tuples are defined with constraints like $t_{=0} \rightarrow (a_{=1} \wedge b_{=2} \wedge c_{=3})$. If, for example, A loses the value 1 from its domain, this constraint will unit propagate, setting $t_{=0}$ to false. This also holds when assignments are guessed: by the core invariant, asserting through RUP that the guessed assignments imply false will propagate the value loss, which in turn propagates the tuple becoming infeasible.

In contrast, suppose we have only two tuples supporting $A = 1$, and these are both made infeasible by other variables, so a solver infers $A \neq 1$. Let $\mathcal{G}$ be the set of equality variables corresponding to our current set of guessed assignments (for example, $\{b_{=2}, c_{=5}\}$). Then the assignment $\overline{a}_{=1}$ does *not* follow by unit propagation in the proof under the assertion of $\wedge\mathcal{G}$, which means it must be justified in some way. Fortunately, this is very simple, and we need only give a small hint to the proof verifier: we claim that $\wedge\mathcal{G} \rightarrow \overline{a}_{=1}$ will be a RUP constraint. Indeed, its negation is $(\wedge\mathcal{G}) \wedge a_{=1}$. Now consider each tuple t supporting $a_{=1}$ in turn. There must be some constraint derived that, under $\wedge\mathcal{G}$, falsifies a different variable in this tuple, which in turn forces the tuple selector variable to not equal t. Additionally, we know that A must take at most one value, and so for each $i \neq 1$, $a_{=i}$ will propagate to false; this in turn propagates every other tuple selector variable to false. Finally, we know that the tuple selector variable must take at least one value, but we have ruled out every value it could take, giving the desired contradiction.

Putting these facts together, the only proof logging needed for a table constraint is to log one RUP step whenever a variable loses a value due to lack of support. Intuitively, infeasibility of tuples is so obvious that we need not mention it. In contrast, loss of support is not immediately obvious to detect, but if we tell the proof verifier that it has in fact occurred then it is easy to check that we are telling the truth.

4.2 Explicit Justifications and Integer Linear Inequalities

Some constraints require more work. Elffers et al. [10] have already shown how both propagation and failure detection for the all-different constraint can be justified using cutting planes proofs. Their approach works in our setting, with one caveat: they require at-least-one constraints for certain CP variables, which we do not have in our model. However, recall that these constraints can be introduced using RUP where needed.

Integer linear inequalities are a similar case. Suppose we have a constraint $2A+3B+4C \leq 42$, with all three variables non-negative. In a typical CP solver, this constraint will achieve integer bounds consistency [17, 4]. As an example, suppose we know that under some set of guessed assignments $\mathcal{G}$ that $A \geq 5$ and $C \geq 3$, then a CP solver will infer that $\wedge\mathcal{G} \rightarrow B \leq 6$. We can derive this fact in a proof as follows. By assumption, we either have or can introduce RUP constraints that $\wedge\mathcal{G} \rightarrow a_{\geq 5}$ and $\wedge\mathcal{G} \rightarrow c_{\geq 3}$. This in turn means we either have or can introduce RUP constraints for the binary representation, saying that $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h-1} 2^i a_{\mathrm{b}i} \geq 5$ and $\wedge\mathcal{G} \rightarrow \sum_{i=0}^{h'-1} 2^i c_{\mathrm{b}i} \geq 3$ for appropriate values of h and h'. Now using cutting planes steps, we can multiply the first of these by 2 and the second by 4 (their coefficients in the original linear inequality), add both to the constraint defining the linear inequality, and divide the result by the coefficient of B, 3. It can be verified that the resulting mess is now sufficient to make $\wedge\mathcal{G} \rightarrow \overline{b}_{\geq 7}$ a RUP constraint. It is also routine to prove that this example generalises to arbitrary integer linear inequalities (although negative variables and / or coefficients require several awkward case by case analyses).

4.3 Element Constraints

Recall the special case of a two-dimensional element from constants constraint, where a variable V takes the $[I, J]$th entry in a two dimensional $m \times n$ array A of constants. In the interests of having a simple pseudo-Boolean encoding, we defined this using $(i_{=x} \wedge j_{=y}) \rightarrow v_{=A[x,y]}$ constraints. However, we wish for our solver to achieve generalised arc consistency on I and J, and bounds consistency on V. One way to proof log this reasoning is as follows. As a one-time operation at the start of search, we will use extension variables to turn this into a one-dimensional element constraint. We will introduce $m \times n$ new variables s_k, each of which is true if and only if a different $i_{=x} \wedge j_{=y}$ holds. We will then build an at-least-one constraint over the s_k variables via a sequence of $O(m \times n)$ RUP steps, as follows. For each value x for the first index variable I, we are going to derive via RUP that $\sum_k s_k + \overline{i}_{=x} \geq 1$. For this to hold, we first derive via RUP that for each value y for the second index variable J, that $\sum_k s_k + \overline{i}_{=x} + \overline{j}_{=y} \geq 1$. The desired at-least-one constraint now follows via RUP; in effect, we have performed an exhaustive backtracking search over the pair of variables I and J under the assertion that the desired at-least-one constraint does not hold, and shown that no solution satisfying I and J exists. From this point forwards, we are effectively dealing with a one-dimensional element constraint.

(Of course, one could ask why we convert from two dimensions to one dimension in the proof, and not in the model when we define the element constraint. We could certainly do things this way. However, our point here is to demonstrate that *we don't have to* handle model reformulations by changing the model: instead, we can use the most straightforward low-level model imaginable, and then prove that our reformulations are valid as part of the proof. We will explore this further below.)

We can also view our new encoding as being like a table constraint, but where the tuple selector variable is channeled to the two index variables. If we wished to achieve generalised arc consistency on the assigned variable V, we would simply reuse the inference techniques discussed in Section 4.1. However, this would require introducing a pseudo-Boolean equality variable $v_{=n}$ for each value in V's domain. This is potentially not what is desired, if the range of the constants is large and V is only otherwise used in a bounds-consistent manner. Therefore, instead of justifying that V does not take each value no longer present in feasible parts of A in turn, we would like to assert a bounds change using a $v_{\geq n}$ variable. This does not immediately follow by RUP on its own, although it will if we repeat the iteration technique used in creating the index variable, but iterating only over feasible array entries.

A downside to this approach is that it produces a proof containing $O(m \times n)$ steps to justify each bounds propagation. As Michel et al. [24] explain, by storing the array in a sorted manner, it is possible for a propagator to avoid looking at most array entries most of the time, and so have better than $O(m \times n)$ performance in the typical case. We suspect that this algorithm can be replicated in a proof efficiently, if we are prepared to establish a set of ordering constraints at the start of the proof.

Finally, an observant reader might have noticed that deletions on the one-dimensional array index will not unit propagate backwards to I and J. In fact, these deletions are RUP.

4.4 Not Equals

At this point, it should be clear how the not equals constraint can be handled: when one variable A is instantiated to a value v, it follows using RUP that the other variable B cannot also be v since the flag f would have to be both true and false to allow $f \rightarrow (A - B > 0)$ and $\overline{f} \rightarrow (B - A > 0)$ to hold simultaneously. However, there is another alternative, which we

will see is more efficient in some scenarios. Instead of deriving under a sequence of guesses $\mathcal{G}$ that $\wedge\mathcal{G} \vee a_{=v} \rightarrow \overline{b}_{=v}$, we could simply introduce the RUP constraint $\overline{a}_{=v} + \overline{b}_{=v} \geq 1$, independently of the guesses. Propagation of the not equals constraint for v would then follow by simple unit propagation.

4.5 Autotabulation and Other Reformulations

Linear equality constraints can be defined and propagated as two linear inequalities. However, sometimes a solver may wish to achieve generalised arc consistency on a linear equality. This is NP-hard, but is still a good idea sometimes for small variables—for example, if $2X + 2Y + Z = 7$ then Z must be odd, but this will not be inferred from the inequalities. One way a solver might handle such constraints is by automatically turning the two linear inequalities into a table constraint. An implementation of this process might, of course, be buggy, and so we would like to prove that this tabulation is valid, rather than simply defining the table constraint in the pseudo-Boolean model. This is indeed possible, using a more advanced form of the kind of argument previously used to turn a 2D element constraint into a 1D element constraint.

Let us start by finding the set of solutions to the constraint. For each solution, we introduce an extension variable t_s which is true if and only if that solution is selected, in the same way as for a table constraint. We also introduce an extension variable g which is true if and only if at least one of these t_s variables is true. Next we perform and log a backtracking search to find all of the solutions to the constraint, except that we use g as an additional guessed assignment at every stage. At the end of the search, we have a proof that g must be true, which in turn gives us an at-least-one constraint over the t_s variables. We have now created all the constraints we need to define a table constraint.

We expect that similar techniques will be useful for many other kinds of reformulation as well, re-emphasising our ability to prove more than just the core solving process. One modelling technique that likely *cannot* easily be handled this way is symmetry breaking constraints. However, Bogaerts et al. [2] show that a slight extension to the VeriPB proof system would make this possible: this raises the intriguing possibility of taking a symmetry breaking lex or ordering constraint that is defined in a high level model, omitting it from the pseudo-Boolean model, and then efficiently proving that the constraint is in fact valid.

5 An Implementation

We have implemented a basic constraint programming solver which supports proof logging (see supplementary material). Our solver generally follows a conventional design, similar to MiniCP [24], although we have chosen for novelty reasons to make use of some modern C++ features like lambdas and variant types instead of a pure object-oriented design. We were not aiming for sheer speed, and so our solver does not include optimisations like multiple propagation queues, backtrackable variables, stateful or support-tracking propagators, or special handling of binary variables. The solver supports only integer variables, and implements the absolute value, all different, comparison (with half and full reification), element, linear equality and inequality, minimum and maximum, and table constraints. We include example programs implementing four of the five MiniCP benchmarks (a quadratic assignment problem, n-queens, magic series, and magic square; the TSP example is not included because

we do not yet have a circuit constraint), as well as the MiniZinc cake optimisation problem[2], the classic "send more money" and Crystal Maze problems, the world's hardest Sudoku puzzle[3], and an odd-even sum problem using an auto-tabulated GAC sum constraint.

Throughout the development process, we have *not* tried particularly hard to produce a solver which is free from bugs. Instead, our goal is to produce a solver that will not produce undetectable incorrect outputs. The rest of this section describes the key aspects of the solver design that involve proof logging, discusses what we have learned from using the VeriPB system in a constraint programming setting, and evaluates its performance.

5.1 Constraint Compilation, or Why Trust the OPB File?

To create the OPB file, we use a single pass approach, outputting definitions as soon as variables and constraints are generated. Variables are handled centrally, whilst each constraint is responsible for providing its own pseudo-Boolean encoding. OPB creation is done purely using text, and the solver stores only the model line numbers for certain constraints—it does not explicitly store any pseudo-Boolean information.

An obvious difficulty with our proposed process is that this compilation from a CP model to an OPB model is not verified. This is somewhat offset by the deliberate use of extremely simple encodings, but one must ask: "why are the authors so sure that they have designed and implemented the encoding correctly, particularly for fiddly global constraints?". The answer to this question is that we are not sure at all, and so we rely upon a special test system, as follows. For a given constraint, we generate many different possible input domains for its variables. For each set of inputs, we use a small generate-and-test program that provides the full set of solutions to the constraint, making no use of the constraint programming solver or any clever logic or programming. (This can be moderately slow, for example for the element constraint.) We then use the constraint solver to solve the problem consisting of just that constraint on those inputs, and verify that the set of solutions found this way is identical. Finally, we verify the proof produced by this solving process: because this is an enumeration problem, this verifies that the OPB file also has exactly the same set of solutions (and additionally that the propagator found them all legitimately, although this is not the main point of the test).

This process is not perfect, but it does severely reduce the scope for errors: for example, it immediately flagged a typo where a reified greater than or equal constraint had accidentally been implemented as a reified greater than constraint, and a bug when the index constraint for an element constraint contained only out-of-range values.

5.2 Producing the Proofs

Recall that to produce a proof, we need to log our backtracking search, and certain variable-value eliminations. In the design of our solver, we opted for a careful separation of the notion of a variable and its current state: the former we represent as a handle, whilst the latter is stored separately in a central location to allow for easy backtracking. It was therefore natural to force every modification to a variable's values to go through a common set of functions, and to make these functions take a mandatory argument that can be either "no justification needed", "output a RUP statement for this", or "call the following piece of code to produce an explicit set of proof steps". Making this argument mandatory forces constraint authors to

[2] `https://www.minizinc.org/doc-2.5.5/en/modelling.html#an-arithmetic-optimisation-example`
[3] `https://abcnews.go.com/blogs/headlines/2012/06/can-you-solve-the-hardest-ever-sudoku`

think explicitly about justifications, and avoids the potential for illicit modifications to be hiding in places where they can not easily be found by inspecting a proof log.

For backtracking search, we treat guessing on a branching variable to be a special kind of inference. Outputting the proof log then simply consists of tracing the search as backtracks are performed. Again, at no point was it necessary to manipulate pseudo-Boolean constraints or proof steps as anything other than simple strings created using a template.

5.3 Identifying Solver Bugs

Our experience has been that once the core solver is working and producing proofs for simple problems, it is somewhat more common to have bugs in the proof-producing code for new propagators than in the propagators themselves. Usually these bugs are extremely easy to fix, because VeriPB immediately flags the faulty line of the proof, and our solver can include a comment line immediately above any proof line saying exactly where in its source code that line originated. Similarly, propagator bugs are usually obvious from proof logs. For example, when we first implemented propagation for linear inequalities, we did not yet have a full proof logging setup for variables with large domains. We therefore used a VeriPB feature which allows for unchecked assertions to be included in the proof log (subject to an angry warning being issued at the end of the verification process) so that the remainder of the proofs could be verified. However, our implementation contained a bug, because one of the authors did not realise they did not understand the rules for rounding and integer division when both a variable and its coefficient are negative. Throughout conventional testing on the remainder of the solver, we never saw a single wrong answer being produced by this bug—but as soon as proof logging was implemented, we were told the exact line of code in our solver that was incorrect, even though correct sets of solutions were still being produced. Of course, one could claim that better testing would have identified this, but this relies upon the tester having intimate knowledge of how the propagator works and remembering that integer division of negative numbers could be a source of errors.

It can sometimes be harder to understand the problem when faulty proofs arise from insufficient justifications. For example, for the absolute value constraint, one of the authors had originally lazily assumed that its propagations would follow by a single RUP step in the same way as for not equals—and indeed this is often but not quite always the case. (This experience has left us extremely envious of the skills of authors of lazy clause generation solvers, who are able to write similar propagators without the benefit of machine verification.) This can lead to a proof verification error that only occurs several propagation steps later than the actual bug: the verifier always tells us if something is wrong, but does not always make it trivial to figure out where. However, because our solver forces all propagations to go through a central function call, it is easy to change the way proof logs are written so that all propagations are checked, including those which would usually be implicit.

5.4 Performance and Overheads

Having discussed the design and implementation of proof logging, we now talk about actually using it. This section answers two questions: “does proof logging work at all?”, and “how expensive is proof logging in practice when used on large problems?”.

To answer the second question, we must first establish whether our solver is “fast enough” that its results are likely representative of what would be seen if proof logging were introduced into a mature solver. For MiniCP, Michel et al. [24] include five benchmarks that are designed to test solver speed: they specify an exact search order, and propagation strength for global

QAP: a quadratic assignment optimisation problem with linear inequalities, not equal constraints, a 2D element constraint, and large variables.						
Runtimes:	MiniCP:	16.9s	OscaR:	7.1s	Choco:	11.3s
	Anon:	5.6s	logging:	149.5s	VeriPB:	232,655.1s
Statistics:	propagators:	355	recursions:	125,805	inferences:	4,521,801
	OPB size:	6.4MBytes	log size:	19GBytes		
	RUP steps:	39,170,568	RPN steps:	413,295	red steps:	101,394
Magic Series: finding the only magic series of length 300, and proving it is unique. Uses linear equality and reified equality constraints.						
Runtimes:	MiniCP:	29.6s	OscaR:	8.8s	Choco:	29.8s
	Anon:	8.2s	logging:	425.2s	VeriPB:	est. 39 days
Statistics:	propagators:	90,301	recursions:	1,193	inferences:	15,584,073
	OPB size:	108MBytes	log size:	12GBytes		
	RUP steps:	7,923,342	RPN steps:	342,401	red steps:	358,800
Magic Square: finding the first 10,000 magic squares of size 5. Uses sum, not equal, and less than constraints.						
Runtimes:	MiniCP:	61.1s	OscaR:	32.3s	Choco:	32.9s
	Anon:	31.0s	logging:	1894.1s	VeriPB:	108,772.8s
Statistics:	propagators:	315	recursions:	6,042,079	inferences:	92,891,165
	OPB size:	145KBytes	log size:	100GBytes		
	RUP steps:	141,528,806	RPN steps:	70,946,952	red steps:	2,550
Queens: finding the first solution to the 88 queens problem. Uses not equals constraints.						
Runtimes:	MiniCP:	876.2s	OscaR:	477.8s	Choco:	438.8s
	Anon:	410.0s	logging:	3450.5s	VeriPB:	60,643.7s
Statistics:	propagators:	11,484	recursions:	49,339,390	inferences:	535,852,330
	OPB size:	8.9M	log size:	104GBytes		
	RUP steps:	50,130,687	RPN steps:	0	red steps:	31,152
Crystal Maze on the usual 8-vertex graph, all solutions. Uses GAC all-different, absolute value, and sum constraints.						
Runtimes:	Anon:	0.01s	logging:	0.13s	VeriPB:	6.3s
Statistics:	propagators:	35	recursions:	259	inferences:	8,737
	OPB size:	60K	log size:	2.6MBytes		
	RUP steps:	32,903	RPN steps:	6,685	red steps:	1,496
With autotabulation and GAC propagation on the sum constraints:						
Runtimes:	Anon:	0.01s	logging:	0.06s	VeriPB:	3.9s
Statistics:	propagators:	52	recursions:	139	inferences:	2,601
	OPB size:	60K	log size:	2.0MBytes		
	RUP steps:	29,467	RPN steps:	102	red steps:	2,958
Sudoku on Arto Inkala's "world's hardest Sudoku puzzle", all solutions. Uses GAC all-different and equals constraints.						
Runtimes:	Anon:	0.03s	logging:	0.05s	VeriPB:	0.52s
Statistics:	propagators:	48	recursions:	103	inferences:	1,388
	OPB size:	320K	log size:	484KBytes		
	RUP steps:	4,561	RPN steps:	460	red steps:	0

Table 1 Experimental results from our anonymous solver on six different problem instances. The first four problems are from the MiniCP benchmark suite and have a fixed model, search order, and propagation strength, to allow for a fair comparison between solvers. The final two problems are relatively simple, but use further global constraints that are not supported in MiniCP.

constraints, so that every solver is producing the same search tree for a fair speed comparison. Their aim was not to have the best model or search for a problem, but rather to benchmark solvers performing the same well-defined task. We support enough global constraints (linear inequalities, sum, not equals, reified equals, a special element constraint, and less than) to implement four of these five benchmarks; we do not yet support the circuit global constraint for the fifth. In the first four rows of Table 1 we present computational results from a machine with dual Intel Xeon E5-2697A v4 CPUs, 512GBytes RAM, and a pair of solid state drives in a RAID 0 configuration, running Ubuntu 20.04.3 LTS, and benchmarking against the versions of the other solvers included in the supplementary material provided by Michel et al. In each case our solver is faster than the fastest of MiniCP, OscaR, and Choco, although sometimes only by a few percent. We therefore believe that the results that follow cannot be said to be unfairly optimistic due to the use of a slow solver.

Table 1 also shows runtimes for running our solver with proof logging enabled, together with statistics showing the size of the OPB models and VeriPB proof logs produced, and the number of RUP steps, groups of cutting planes steps (VeriPB works with sequences of cutting planes steps in reverse Polish notation, RPN, rather than one step per line), and redundance-based strengthening steps (red; two such steps are used to introduce an extension variable). On the four MiniCP benchmarks we see a slowdown of between 8.4 and 61.1 from proof logging. This should not be particularly surprising: without proof logging, our solver is making between eight hundred thousand and three million successful inferences per second, and the proof logs to justify these inferences range from ten to over a hundred GBytes in size. Furthermore, our implementation of proof logging is deliberately pessimal. We make use of C++ text output streams for file writing, which are notoriously inefficient. We write comments for most proof log lines generated, we make use of expressive variable names (which require several string concatenation operations and a hash table lookup for each variable written out), and proof lines are manipulated as strings for ease of implementation; all of these decisions are extremely helpful for exploratory research, but not for performance. Finally, these MiniCP benchmarks make use of only relatively simple and extremely fast propagators, which is where proof logging is most expensive. We therefore consider these performance results to be close to a worst case scenario, and would not be surprised if the overheads could be cut by at least a factor of five for some problems if implemented in a production solver that aimed purely for performance rather than for research and teaching.

Returning to the first question, we were able to verify the entire proofs for three of the four MiniCP problems; based upon the first ten percent of the proof for the remaining magic series problem, VeriPB estimates it will take 39 days to verify. We were able to verify entire proofs for smaller instances of the magic series problem. We have also produced and verified proofs for a range of other problems that make heavier use of global constraints—we show two of these in the bottom of Table 1, and other example problems and per-constraint tests are included in our supplementary material. Considering these results, and all the bugs that have been identified during development, we are comfortable in claiming that proof logging can be effective in practice. Although it may not (yet) scale practically to some of the more challenging combinatorial benchmark instances, it is already able to handle moderately sized problems involving several different global constraints, large variables, and reformulation.

6 Conclusion and Future Work

Proof logging gives us a way to trust outputs, not solvers. Trusting solvers seem to be a long way from being a practical reality for for constraint programming: even relatively simple

propagators like all different have resisted attempts at formally verified implementation [8] even without the extensive optimisations used by modern solvers [11]. In contrast, we have shown that, with the right proof format, it is relatively easy to add proof logging to a wide variety of propagators, without requiring the proof verifier to understand anything about constraint programming—and this does not stand in the way of propagator optimisations such as greediness and incrementality.

There is still a lot of work to do before proof logging can be used by everyone all of the time. Firstly, there are many more global constraints and propagators to consider. Most of these will be straightforward, and will re-use existing strategies for proof logging in familiar ways. However, some will not be, and it is an open question as to whether cutting planes with extension variables give a sufficiently strong system to provide practical proof logging in every situation. We expect that recent work in proof logging for symmetry and dominance relations [2] might be necessary to justify certain propagators, as well as for reformulations involving symmetries, and would be interested in a deeper investigation into the relationship between constraint programming propagators and proof systems (with the caveat that "this system polynomially simulates natural deduction and so it can do everything" is not a helpful answer unless the polynomial is of very low order and with small constants).

Secondly, we must think about performance. Using formatted text output and string lookups to produce proof logs is useful for development and exploratory purposes, but for a production solver a better approach is probably needed. Verification performance is also a concern, although we have many reasons to be optimistic that this will improve. For example, very small changes to how proofs are written can give a huge improvement to verification times. We discussed two different ways of proof logging the not equals constraint, one of which involved justifying every propagation subject to the current guessed assignments, and the other which produced new clauses to assist unit propagation. On the MiniCP queens benchmark, using the former would have produced a 1.1TByte proof log that would take an estimated 138 days to verify, rather than a 100GByte proof that could be verified in under a day. If we are prepared to put slightly more cleverness into a solver, and abandon the gratuitous use of RUP steps in favour of a little more logic, we expect that proof sizes for some other constraints can be reduced by a similar factor.

An automated tool that performs proof minimisation would also be useful in this respect. Although potentially expensive to run, this would be very useful for auditability where proofs are to be stored, shared, and potentially verified more than once by different people. Such a tool could also provide annotations that would make RUP steps much quicker to verify—such an approach is already used for formally verified verifiers for DRAT, which actually verify a simplified format called LRAT [6].

On the other hand, relatively slow verification is not a fatal flaw. Proof logging is very good at catching solver bugs that will not be detected by conventional testing, even on relatively small instances. Because the same logic and code paths in a solver can be used whether or not proof logging is enabled, it is a useful feature to support even if it is not enabled all of the time. And, of course, many useful problems with serious real-world consequences derive most of their difficulty from the variety of constraints involved, rather than from being close to the limit of what we can solve computationally.

And thirdly, although we have a reasonably good solution for being confident in our translation from a constraint programming model into OPB, we have not discussed the further difficulty of verifying compilation from high level languages like Essence or MiniZinc. Perhaps it would be worth investigating techniques from formally-verified compilers to help with this translation.

References

1 Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Metamorphic testing of constraint solvers. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 727–736. Springer, 2018. doi:10.1007/978-3-319-98334-9_46.

2 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified symmetry and dominance breaking for combinatorial optimisation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, 2022.

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

4 Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*, pages 49–58, 2006.

5 William J. Cook, Collette R. Coullard, and György Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

6 Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2017. doi:10.1007/978-3-319-63046-5_14.

7 Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016. doi:10.1007/978-3-319-44953-1_14.

8 Catherine Dubois. Formally verified constraints solvers: a guided tour (invited talk). CICM. 2020.

9 Leon Eifler and Ambros M. Gleixner. A computational status update for exact rational mixed integer programming. In Mohit Singh and David P. Williamson, editors, *Integer Programming and Combinatorial Optimization - 22nd International Conference, IPCO 2021, Atlanta, GA, USA, May 19-21, 2021, Proceedings*, volume 12707 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2021. doi:10.1007/978-3-030-73879-2_12.

10 Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-Boolean reasoning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1486–1494. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5507.

11 Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, 2008. doi:10.1016/j.artint.2008.10.006.

12 Xavier Gillard, Pierre Schaus, and Yves Deville. SolverCheck: Declarative testing of constraints. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 565–582. Springer, 2019. doi:10.1007/978-3-030-30048-7_33.

13 Stephan Gocht, Ross McBride, Ciaran McCreesh, Jakob Nordström, Patrick Prosser, and James Trimble. Certifying solvers for clique and maximum common (connected) subgraph problems. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 of *Lecture Notes in Computer Science*, pages 338–357. Springer, 2020. doi:10.1007/978-3-030-58475-7_20.

14 Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Subgraph isomorphism meets cutting planes: Solving with certified solutions. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1134–1140. ijcai.org, 2020. doi:10.24963/ijcai.2020/158.

15 Stephan Gocht and Jakob Nordström. Certifying parity reasoning efficiently using pseudo-Boolean proofs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 3768–3777. AAAI Press, 2021. URL: https://ojs.aaai.org/index.php/AAAI/article/view/16494.

16 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference (DATE)*, pages 10886–10891. IEEE Computer Society, 2003.

17 Warwick Harvey and Joachim Schimpf. Bounds consistency techniques for long linear constraints. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems*, pages 39–46, 2002.

18 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013. URL: http://ieeexplore.ieee.org/document/6679408/.

19 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013. doi:10.1007/978-3-642-38574-2_24.

20 Linnea Ingmar and Christian Schulte. Making compact-table compact. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 210–218. Springer, 2018. doi:10.1007/978-3-319-98334-9_14.

21 Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 545:17–18, 2016.

22 Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints An Int. J.*, 16(4):341–371, 2011. doi:10.1007/s10601-011-9107-6.

23 Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011.

24 Laurent D. Michel, Pierre Schaus, and Pascal Van Hentenryck. MiniCP: a lightweight solver for constraint programming. *Math. Program. Comput.*, 13(1):133–184, 2021. doi:10.1007/s12532-020-00190-7.

25 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007. doi:10.1007/978-3-540-74970-7_39.

26 Adrian Rebola-Pardo and Luís Cruz-Filipe. Complete and efficient DRAT proof checking. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8602993.

27 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.

28 Peter J. Stuckey. Certifying optimality in constraint programming, February 2019. Talk at KTH Royal Institute of Technology.

29 Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18):3639–3678, 2007. `doi:10.1016/j.ins.2007.03.030`.

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

31 Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1754`.

32 Hélène Verhaeghe. *The extensional constraint*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2021. URL: `http://hdl.handle.net/2078.1/252859`.

33 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. `doi:10.1007/978-3-319-09284-3\_31`.

Paper G

Subgraph Isomorphism Meets Cutting Planes: Solving With Certified Solutions

Stephan Gocht[1,2] , **Ciaran McCreesh**[3] and **Jakob Nordström**[2,1]
[1]Lund University, Lund, Sweden
[2]University of Copenhagen, Copenhagen, Denmark
[3]University of Glasgow, Glasgow, Scotland
stephan.gocht@cs.lth.se, ciaran.mccreesh@glasgow.ac.uk, jn@di.ku.dk

Abstract

Modern subgraph isomorphism solvers carry out sophisticated reasoning using graph invariants such as degree sequences and path counts. We show that all of this reasoning can be justified compactly using the cutting planes proofs studied in complexity theory. This allows us to extend a state of the art subgraph isomorphism enumeration solver with proof logging support, so that the solutions it outputs may be audited and verified for correctness and completeness by a simple third party tool which knows nothing about graph theory.

1 Introduction

The subgraph isomorphism decision problem is to find a copy of a small "pattern" graph inside a larger "target" graph, or to show that no such copy exists; the enumeration problem is to find all copies. These problems occur in many applications—we refer to Archibald *et al.* [2019] for a partial list. Although the problems are NP- and #P-complete respectively, a series of algorithms based upon constraint programming [Zampelli *et al.*, 2010; Solnon, 2010; Audemard *et al.*, 2014; McCreesh and Prosser, 2015; Archibald *et al.*, 2019] have culminated in a practical way of tacking all but the hardest instances [McCreesh *et al.*, 2018; Solnon, 2019]. These algorithms exploit various combinatorial and graph invariants, such as matchings, degree sequences, and number of paths between vertices, in a bid to reduce the number of combinations which must be considered. As a result, the solvers implementing these algorithms are rather complex, and even after extensive testing it is hard to be convinced that the solvers are definitely free from bugs.

This paper discusses proof logging as a way of verifying the solutions produced by of one of these solvers: the idea is that the solver is modified to produce a "certificate" or proof file as part of its output, which can then be verified by a (much simpler) external tool. For satisfiable decision instances for NP problems, such certificates are always small, and (usually) easy to check. For demonstrating unsatisfiability, or for showing that a solver has not missed any solutions when enumerating, no way of guaranteeing short certificates is known. However, theoretical worst cases are overly pessimistic, and modern subgraph isomorpism solvers often perform much better than exponential worst-case performance bounds would suggest. In the same way, we will show that with the right proof format, certificates can be simple to verify, yet still only be proportional in size to the amount of work carried out by a solver.

We stress that proof logging does not prove that a solver is correct: unless a solver actually exhibits buggy behaviour when producing a proof, a proof verifier will not complain. On the other hand, proof verifiers will detect if a correct solver is run on faulty hardware or is compiled with a buggy compiler, if that leads to the solver performing incorrect reasoning. In other words, proof logging gives us a way of trusting solver *outputs*, not solvers.

Proof logging in the Boolean satisfiability community is usually done using a format known as DRAT [Heule *et al.*, 2013b; Heule *et al.*, 2013a; Wetzler *et al.*, 2014]. Recently, Elffers *et al.* [2020] proposed a different proof-logging format based upon cutting planes proofs for pseudo-Boolean models, and showed that (unlike DRAT) it could easily handle the all-different reasoning used in constraint programming solvers. Because subgraph isomorphism solvers also make use of strong all-different reasoning and similar counting arguments, we will be using this format. Our first contribution is to show that cutting planes proofs are also powerful enough to compactly express reasoning about graph degrees, neighbourhood degree sequences, and counts of short paths in graphs. This is sufficient to represent all of the reasoning carried out by the Glasgow Subgraph Solver [Archibald *et al.*, 2019], which is the current strongest subgraph isomorphism solver on hard instances [Solnon, 2019]. This is a surprising result: cutting planes proofs know nothing about graphs, and the solver's inference algorithms were not designed with proof logging in mind. Our second contribution is to demonstrate that this approach is actually practical: we extend the Glasgow solver with proof logging support, and produce and verify solution certificates for over a thousand standard enumeration benchmark instances.

2 Background

We begin by introducing notation, and providing the necessary background on graphs and on pseudo-Boolean formulae.

Graphs. Let G be a graph with vertex set $\mathrm{V}(G)$, and let $v \in \mathrm{V}(G)$. We write $\mathrm{N}(v)$ for the neighbourhood of (set of

vertices adjacent to) v, and $\deg(v)$ for the cardinality of the neighbourhood; we write $\deg(G)$ for the mean of the degrees of all vertices of G. For simplicity, every graph appearing in this paper is undirected, unlabelled, and does not have loops (vertices adjacent to themselves), although every result we describe can be extended to these more general cases.

Subgraph isomorphism. Given a *pattern* graph P and a *target* graph T, the *non-induced subgraph isomorphism problem* is to find an injective mapping from $\mathrm{V}(P)$ to $\mathrm{V}(T)$ such that adjacent vertices in P are mapped to adjacent vertices in T. The *induced* problem additionally requires that non-adjacent vertices be mapped to non-adjacent vertices—again, we do not discuss this further in this paper, although our results are also easily applicable to this problem. The *enumeration* problem is to find every such mapping. (Some works instead consider the *unlabelled enumeration* variant, defined as finding every *image* of such a mapping.)

Pseudo-Boolean formulae. A pseudo-Boolean (PB) formula consists of a set of $\{0,1\}$-valued variables $\{x_1, \ldots, x_n\}$ together with a set of linear constraints $\sum_{i=1}^{n} a_i \ell_i \geq A$, where each a_i and A is an integer, and each ℓ_i is either a literal x_i or a negated literal $\overline{x}_i$, where $x_i + \overline{x}_i = 1$. We can convert a Boolean satisfiability (SAT) problem instance in conjunctive normal form (CNF) into a PB formula because, e.g. $x_1 \vee x_2 \vee \overline{x}_3$ is satisfied iff $x_1 + x_2 + \overline{x}_3 \geq 1$, but in general the PB format is exponentially more expressive.

Cutting planes proofs. The *cutting planes* (CP) proof system [Cook *et al.*, 1987] allows us to reason about the satisfiability or unsatisfiability of a PB formula, in a similar way to the resolution system for SAT. Briefly, starting with the input constraints, we may generate new constraints by adding existing constraints, multiplying them by an integer constant, dividing by a positive integer constant (with rounding up), and introducing literal constraints $\ell_i \geq 0$. The VeriPB tool [Elffers *et al.*, 2020] provides a way of encoding CP proofs in such a way that they can be verified by machine: we refer to the tool's documentation[1] for details.

Simplification and reverse unit propagation. As well as accepting manual derivations of new constraints from existing ones, VeriPB has two ways of introducing a constraint which is specified explicitly during a proof log. The first is if the new constraint is semantically implied by an existing constraint (that is, if it may be obtained by weakening coefficients and cancelling literals). The second is through *reverse unit propagation* (RUP) [Goldberg and Novikov, 2003; Elffers *et al.*, 2020]: if the negation of the new constraint combined with every existing constraint is "obviously" unsatisfiable through unit propagation, then the new constraint may be added. Note that RUP constraints add no new expressive power and can be relatively expensive for the verifier, but using them appropriately can make solver implementation vastly more straightforward.

Logging of solutions. To support enumeration problems, VeriPB allows solutions to be logged. These are checked as they are encountered, and then their negations are added as new constraints. Thus, a proof log for an enumeration problem is effectively a list of solutions, plus an unsatisfiability proof showing there are no further solutions that were missed.

[1] https://github.com/StephanGocht/VeriPB/

3 Reasoning about Subgraphs

We will now demonstrate that all of the preprocessing and reasoning carried out by the Glasgow subgraph solver can be justified compactly using CP proofs. We will discuss all of the kinds of reasoning carried out by the Glasgow solver, but we will not describe precisely how these different steps fit together to make an algorithm—we refer to McCreesh and Prosser [2015] and Archibald *et al.* [2019] for those details. We will also touch upon kinds of reasoning carried out by other subgraph solvers, showing the generality and limitations of these results. However, because the VeriPB tool only understands PB formulae, we must first explain how we encode a subgraph isomorphism problem as a PB formula.

3.1 A Pseudo-Boolean Encoding

In the common constraint programming encoding for subgraph isomorphism used by the Glasgow solver, we have a variable for each vertex in the pattern graph, and each domain ranges over the vertices of the target graph. In other words, we are building a mapping from the pattern graph to the target graph. In a pseudo-Boolean model, we replace each constraint programming variable with a set of Boolean variables, one for each value in its domain—each of these variables $x_{p,t}$ represents a pair consisting of a pattern vertex p and a target vertex t, and is set to true precisely if p is to be mapped to t. Conveniently for the enumeration problem, solutions to this PB formula will be in one-to-one correspondence with solutions to the actual problem.

Our first set of constraints says that each pattern vertex must be mapped to exactly one target vertex:

$$\sum_{t \in \mathrm{V}(T)} x_{p,t} \geq 1 \qquad p \in \mathrm{V}(P)$$

$$\sum_{t \in \mathrm{V}(T)} -x_{p,t} \geq -1 \qquad p \in \mathrm{V}(P)$$

We then express injectivity, by saying that each target vertex may be used at most once:

$$\sum_{p \in \mathrm{V}(P)} -x_{p,t} \geq -1 \qquad t \in \mathrm{V}(T)$$

Finally, we must express the adjacency constraints. The most obvious way to do this is by saying that edges cannot be mapped to non-edges:

$$-x_{p,t} + -x_{q,u} \geq -1 \qquad p \in \mathrm{V}(P),\ q \in \mathrm{N}(p),\ t \in \mathrm{V}(T),\ u \in \mathrm{V}(T) \setminus \mathrm{N}(t)$$

However, it is more convenient and compact (particularly if the target graph is sparse) to reformulate this by saying that if a vertex p is mapped to a vertex t, then every vertex in the neighbourhood of p must be mapped to a vertex in the neighbourhood of t:

$$\overline{x}_{p,t} + \sum_{u \in \mathrm{N}(t)} x_{q,u} \geq 1 \quad p \in \mathrm{V}(P),\ q \in \mathrm{N}(p),\ t \in \mathrm{V}(T)$$

All of these constraints must then be expressed in the OPB file format[2]. For readability, VeriPB allows us to name $x_{p,t}$ variables with strings like `xp_t`, whereas many PB solvers accept only numerical variable names like `x123`. Because the encoding process is not verifiable and the verifier cannot detect bugs in the encoding process, we keep the encoding as simple as possible and do not perform any reasoning here.

Note that this encoding has $O(|\,\mathrm{V}(P)|\deg(P)|\,\mathrm{V}(T)|)$ constraints, and that each adjacency constraint is potentially $O(|\,\mathrm{V}(T)|)$ long (although for sparse target graphs the length will be considerably shorter). The Glasgow solver instead represents the constraints using two bitset matrices, requiring only $O(|\,\mathrm{V}(P)|^2 + |\,\mathrm{V}(T)|^2)$ size, whilst some other solvers use even smaller adjacency list representations. This does provide us with a fairly moderate limit on the size of graphs with which we may use this verification process, compared to what subgraph solvers can handle. This is also one of the reasons that feeding such an input to a pseudo-Boolean (or Boolean satisfiability) solver is not a particularly good way of solving the problem in practice: dedicated subgraph isomorphism solvers give *much* better performance. However, the proof logs we will produce will correspond to a sequence of steps which *could*, in theory, have been carried out by a pseudo-Boolean solver working on these models.

3.2 Adjacency and Backtracking Search

Elffers *et al.* [2020] described how the recursive calls carried out by a standard backtracking constraint programming search algorithm could be logged using reverse unit propagation constraints, requiring one RUP constraint for every backtrack. They point out that with this approach, there is no need to log any inference steps carried out by a logging solver if they are no stronger than unit propagation on the associated OPB model. The Glasgow solver only performs inference on adjacency constraints when it is decided that a specific pattern vertex must be mapped to a specific target vertex, and so the following proposition is immediate.

Proposition 1. PB unit propagation on adjacency constraints carries out the same reasoning as the Glasgow solver, and so requires no explicit logging when using RUP.

This result, combined with a limited application of all-different justification [Elffers *et al.*, 2020], is already enough to deal with simple-but-fast subgraph isomorphism solvers like RI [Bonnici *et al.*, 2013] and VF2 [Cordella *et al.*, 2004] which do not use constraint programming techniques and which do not perform any further strong inference during search. To log the behaviour of "cleverer" constraint programming style algorithms like the Glasgow solver, however, we need to be able to justify several other kinds of preprocessing and reasoning. In the same way that Elffers *et al.* [2020] produced proofs by combining RUP with additional explicitly-derived constraints for verifying all-different reasoning in a constraint programming solver, we next show how to provide the verifier with enough additional information that every variable-value deletion in the subgraph solver will be reflected in the PB representation following RUP.

[2]http://www.cril.univ-artois.fr/PB12/format.pdf

3.3 Reasoning About Degrees

A pattern vertex p of degree $\deg(p)$ can never be mapped to a target vertex t of degree $\deg(p) - 1$ or lower in any subgraph isomorphism. Expressing this fact using resolution proofs would require exponential length [Haken, 1985], but in cutting planes a proof may easily be derived. We demonstrate this by example. Suppose $\mathrm{N}(p) = \{q, r, s\}$ and $\mathrm{N}(t) = \{u, v\}$ for some pattern vertex p and target vertex t: we wish to derive $\overline{x}_{p,t} \geq 1$. We start with the three adjacency constraints,

$$\begin{aligned}\overline{x}_{p,t} + x_{q,u} + x_{q,v} &\geq 1\\ \overline{x}_{p,t} + x_{r,u} + x_{r,v} &\geq 1\\ \overline{x}_{p,t} + x_{s,u} + x_{s,v} &\geq 1,\end{aligned}$$

whose sum is

$$3\overline{x}_{p,t} + x_{q,u} + x_{q,v} + x_{r,u} + x_{r,v} + x_{s,u} + x_{s,v} \geq 3.$$

Observe that due to injectivity, at most one of the column $x_{q,u}$, $x_{r,u}$, and $x_{s,u}$ can be true, and similarly for the column of $x_{-,v}$ variables. Adding the injectivity constraints for target vertices u and v to the sum of the adjacency constraints gives

$$3\overline{x}_{p,t} + \sum_{p\in\mathrm{V}(P)\setminus\{q,r,s\}} -x_{p,u} + \sum_{p\in\mathrm{V}(P)\setminus\{q,r,s\}} -x_{p,v} \geq 1,$$

which is *almost* what we want except that we have acquired some additional variables from the injectivity constraints. This is not a problem: we can remove these stray $x_{p,-}$ variables by adding literal axioms (since $x_i \geq 0$ for any variable x_i) and then finally divide the resulting expression by 3, to obtain $\overline{x}_{p,t} \geq 1$ as desired. In proof logging terms, this whole process can be expressed in a single "p" ("reverse Polish expression") rule in the VeriPB format, optionally followed by an "e" ("equals") rule for sanity-checking purposes. With added line breaks and comments, this could look like:

```
p 18 19 + 20 +      * sum adj constraints
  12 + 13 +         * sum inj constraints
  xp_u + xp_v +     * cancel stray xp_*
  xo_u + xo_v +     * cancel stray xo_*
  3 d 0             * divide, and we're done
e 74 1 ~xp_t >= 1 ; * check what we just did
```

Alternatively, because the desired constraint only using weakening of coefficients and cancellation of literals, we may use a "j" ("implies and add") rule to avoid listing the steps explicitly:

```
p 18 19 + 20 +      * sum adj constraints
  12 + 13 + 0       * sum inj constraints
j 74 1 ~xp_t >= 1 ; * and simplify the above
```

In general, following the above process can justify *any* degree reasoning step:

Proposition 2. If a pattern vertex p cannot be mapped to a target vertex t due to degree, then we can justify this using a single "p" rule containing $\deg(p)+\deg(t)$ additions of model axioms, and a single "j" rule.

The Glasgow solver does not just reason about degrees, though: it also reasons about *global* and *neighbourhood degree sequences*, using a result due to Zampelli *et al.* [2010].

Let $\mathrm{S}(v)$ be the sequence consisting of the degrees of the neighbours of vertex v, from largest to smallest. Then a pattern vertex p can only be mapped to a target vertex t if $\mathrm{S}(p)$ is pointwise less than or equal to $\mathrm{S}(t)$. Similarly, if the sorted global degree sequence of the pattern graph as a whole is not pointwise less than or equal to the sorted global degree sequence of the target graph, then the problem is unsatisfiable.

Proposition 3. We may justify unsatisfiability due to global degree sequence reasoning using no more than $O(|\,\mathrm{V}(P)| + |\,\mathrm{V}(T)|)$ extra steps, following degree reasoning.

To do this, let i be the position of the first mismatch of the sequence. We first perform degree reasoning to eliminate the ith and all subsequent lower degree target vertices from the first i pattern vertices in the sequence. Then the first i pattern vertices and the first $i-1$ target vertices in the sequence form a Hall violator, which may be used to demonstrate unsatisfiability following the process described by Elffers *et al.* [2020].

Neighbourhood degree sequence reasoning is also expressible as a CP proof—we will demonstrate by example. Suppose a pattern vertex p has neighbourhood degree sequence $(5, 4, 3, 1)$ from neighbours (q, r, s, o), and a target vertex t has neighbourhood degree sequence $(5, 4, 2, 2)$ from vertices (u, v, w, x). In this case, the third item in the degree sequence is the first mismatch, so we will sum up the adjacency constraints for the first three pattern vertices to get

$$\begin{aligned} 3\overline{x}_{p,t} \quad & +x_{q,u} + x_{q,v} + x_{q,w} + x_{q,x} \\ & +x_{r,u} + x_{r,v} + x_{r,w} + x_{q,x} \\ & +x_{s,u} + x_{s,v} + x_{s,w} + x_{s,x} \qquad \geq 3. \end{aligned}$$

Now observe that, because the mismatch starts at the third item in the sequence, the third and subsequent columns of $x_{-,w}$ and $x_{-,x}$ variables all correspond to assignments which are impossible due to degree. We may therefore remove these variables by adding in the $\overline{x}_{-,w} \geq 1$ clauses created using the steps in the previous subsection. This leaves us with

$$3\overline{x}_{p,t} + x_{q,u} + x_{q,v} + x_{r,u} + x_{r,v} + x_{s,u} + x_{s,v} \geq 3.$$

At this point, we are in a very similar situation to with degree reasoning, above: the $x_{-,u}$ and $x_{-,v}$ sets of variables can both contribute at most one to the sum, due to injectivity. So, we add the injectivity constraints as before, and then either explicitly eliminate stray variables and divide, or simply ask the proof verifier to derive the exact constraint by implication from this sum. By generalising this example, we can conclude the following proposition.

Proposition 4. If a pattern vertex p cannot be mapped to a target vertex t due to neighbourhood degree sequence, then we can justify this using a single "p" rule containing no more than $\deg(p)+\deg(t)$ additions of model axioms, and no more than $\deg(t)|\,\mathrm{V}(P)|$ additions of previously derived rules, followed by a single "j" rule for simplification.

Zampelli *et al.* [2010] also make use of *dynamic* degree sequences in their solver: if a target vertex no longer appears in the domain of any pattern vertex (either initially, or dynamically inside search), then it is considered deleted and degrees and degree sequences are recalculated. The Glasgow solver does not use this inference, but if it did it would not be a problem for proof logging, as we would simply add in the derived constraints showing that no pattern vertex can be mapped to the target vertex in question. In the same way as for the all-different constraint, reverse unit propagation will automatically handle the current set of guessed assignments.

3.4 Reasoning About Paths

Audemard *et al.* [2014] implemented a solver named SND which propagated based upon distances as well as adjacency: if the distance between two pattern vertices p and q is d, and they are mapped to target vertices t and u respectively, then the distance between t and u must be no more than d. This filtering was refined in an early iteration of the Glasgow solver [McCreesh and Prosser, 2015] and in the PathLAD solver [Kotthoff *et al.*, 2016] as follows: call two vertices v and w $[k,d]$-*adjacent* if they have at least k simple paths of *exactly* length d between them. Then if p and q are $[k,d]$-adjacent for any k and d, then t and u must also be $[k,d]$-adjacent. This form of filtering is extremely expensive computationally if d and k are arbitrary, so the current version of the Glasgow subgraph solver uses only $d = 2$ and $k \leq 4$.

Instead of using path counts directly for filtering, the Glasgow solver generates additional sets of graph pairs $P^{[k,d]}$ and $T^{[k,d]}$, which have the same vertex sets but with vertices v and w adjacent in $G^{[k,d]}$ if they are $[k,d]$-adjacent in G. The solver then uses adjacency, degree, and degree sequence reasoning over all of these graph pairs, in a way which requires the full strength of the following proposition.

Proposition 5. For fixed k, for every pair of vertices p and q that are $[k,2]$-adjacent in P, and for every target vertex t, PB reasoning can derive in polynomial length a new constraint in *exactly* the form $\overline{x}_{p,t} + \sum_u x_{q,u} \geq 1$, where the u sum ranges over vertices that are $[k,2]$-adjacent to t.

This process is somewhat intricate, so we give only a sketch of how it works. First we establish that if p maps to t, then q maps to a vertex which is a walk of length two away from t, by summing each (p, r, t) adjacency constraint for r in $\mathrm{N}(p) \cap \mathrm{N}(q)$. We then resolve this with each (r, q, u) adjacency constraint in turn for each $u \in \mathrm{N}(t)$, and simplify. We then use injectivity and a second simplification step to strengthen the generated constraint to paths of length two. This requires two expressions with $O(|\,\mathrm{V}(P)||\,\mathrm{V}(T)|)$ terms, and two semantic implication steps.

Finally, for $k > 1$, we must cancel out any $x_{q,u}$ which has insufficiently many paths of length exactly two between it and t. For each such item in turn, we use a simple counting and injectivity argument over the set of potential target vertices for each r to generate a binary clause $\overline{x}_{p,t} + \overline{x}_{q,u} \geq 1$. These are all then added to the original constraint. This requires a further $O(|\,\mathrm{V}(T)|)$ expressions of size $O(|\,\mathrm{V}(P)| + |\,\mathrm{V}(T)|)$, and $O(|\,\mathrm{V}(T)|)$ simplifications.

We suspect it is also possible to use PB reasoning to justify arbitrary-length distance filtering in polynomial length. However, the short exact path count filtering used in the Glasgow solver appears to be both more efficient and more powerful in practice, and no solver since SND has used arbitrary distance filtering.

3.5 Other Algorithmic Features

There are four other core algorithmic features of the Glasgow solver. The first is all-different propagation, which is used as a powerful way of reasoning about injectivity. The Glasgow solver uses a bit-parallel all-different propagator, rather than the usual generalised arc consistency propagator—however, it still performs deletions and backtracking based upon Hall sets, and so the approach described by Elffers *et al.* [2020] for justifying the all-different constraint may be used with no additional work required.

The second feature is restarts with nogood recording. Archibald *et al.* [2019] showed that rather than performing a simple backtracking search, it is better to repeatedly perform a small amount of search and then restart the solver with a new branching strategy. At every restart, a set of nogoods [Lecoutre *et al.*, 2007; Lee *et al.*, 2016] is recorded, so that the solver does not duplicate work it has already carried out. These nogoods are expressed as CNF clauses and are propagated internally using unit propagation, which means they can simply be logged as-is in the proof file.

The third is that if the input graph is a clique, the solver switches to an entirely different algorithm to solve the problem. Implementing proof logging for this second algorithm is a work in progress. The fourth is parallel search. Our experiments will show that proof logging is heavily I/O bound, and parallelism would make this worse.

Other constraint programming inspired subgraph isomorphism solvers make use of features like arc consistency, and all different filtering on edges. Although not discussed here, these features are also justifiable in polynomial length.

4 Implementation and Evaluation

We implemented[3] the proof logging techniques described above in the Glasgow Subgraph Solver; we also used VeriPB's support for deletion of intermediate and temporary constraints, which cut down on verifier memory usage. Critically, we were able to do all this *without making any changes to the core functions or data structures of the solver*, beyond adding in extra optional calls to the proof logging routines: all of the information needed was already either present or easily accessible from within the solver. (This would not have been the case if we could not use RUP constraints.)

Evaluation. There are currently no other proof logging subgraph isomorphism solvers, so we cannot compare our technique to another solver. However, we can demonstrate that the techniques we have described can be implemented, and that producing and verifying subgraph isomorphism proofs works in practice, at least on smaller graphs.

Hardware setup. Our experiments are performed on a cluster of machines with dual Intel Xeon E5-2697A v4 CPUs and 512GBytes RAM, running Ubuntu 18.04. The performance measurements for writing the proof logs are largely governed by hard disk speed, not CPU overheads, and our machines are all equipped with a single conventional hard disk which limits write speeds to around 100MBytes/s. We therefore do not expect our logging times to be reproducible.

[3] https://github.com/ciaranm/glasgow-subgraph-solver

Instances. We use the instances collected by Kotthoff *et al.* [2016] for evaluation. This is a mix of real-world and randomly generated instances, of a varying range of difficulties. Some of the instances are very large, and so even generating OPB files for them would be infeasible. We therefore select every instance where the target graph has no more than 260 vertices, and where the unmodified Glasgow solver without proof logging can enumerate every solution in no more than ten seconds. (We focus on the enumeration problem because it is more of a stress test than proof logging for decision instances would be.) This gives us a total of 1,227 instances, 789 of which are unsatisfiable, with the remainder having somewhere between one and 50,635,140 solutions; 498 of the instances were solved without any guessing, whilst the hardest solved satisfiable and unsatisfiable instances required 53,605,482 and 2,074,386 recursive calls respectively.

Successful results. Our main result is that the technique works. For all but five of these 1,227 instances, we were able to produce proofs and verify their correctness. For the remaining five instances, the verifier took over three days to run (without yet having found any mistakes). Each of these instances were small satisfiable instances with very many (between fourteen and fifty million) solutions, requiring more than twenty million recursive calls to solve, and with proof log files of between twenty and fifty GBytes.

Time costs of proof logging and verification. In the top row of Figure 1 we show the time costs of performing proof logging and verification on these instances. The first plot shows the cumulative number of instances solved over time without proof logging, with proof logging enabled, and for proof verification. The plot suggests a four orders of magnitude slowdown in aggregate for easy instances, dropping to two orders of magnitude for harder instances. Meanwhile, verifying proofs is approximately one order of magnitude slower than producing them. The second plot shows how much slower producing proofs is on an instance by instance basis—we discuss this further below. The third plot shows how many times harder it is to verify a proof than it is to produce it, and shows a close linear correlation.

OPB and proof log sizes. The second row of Figure 1 looks at the size of the generated OPB and proof log files. The largest OPB file (bearing in mind our pre-selection of small instances) is 425MBytes, for a pattern graph with 121 vertices and a target graph with 128 vertices. Meanwhile, some of our proof logs reached many tens of GBytes—although this sounds large, recall that the subgraph solver can carry out over fifty million backtracks within ten seconds.

Where the costs come from. Although not ideal, the slowdowns to the solver from proof logging are to be expected for two reasons. Firstly, the Glasgow solver employs bit parallelism and other algorithmic techniques and data structures designed to allow it to carry out inference extremely quickly. When working with relatively small target graphs, it is able to carry out a full round of inference, variable selection, and recursion in under 0.2 microseconds. If each such round requires 1KByte of logging, we would need to be able to write to disk at around 5GBytes per second to keep up—this is already a factor of fifty higher than what our hardware is capa-

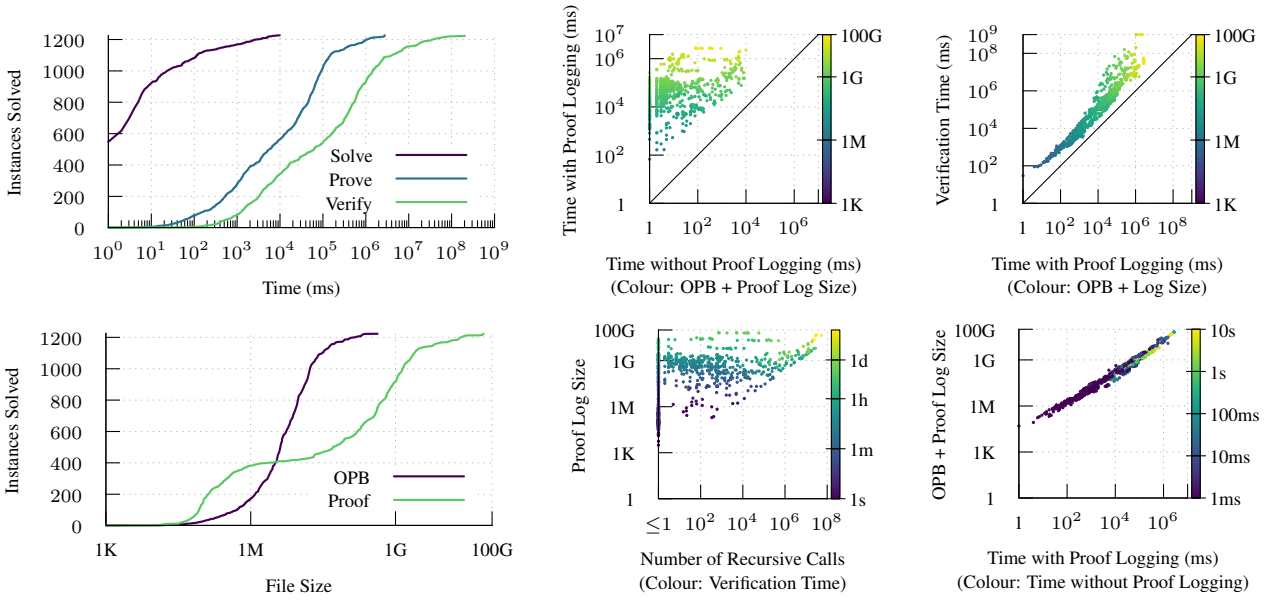


Figure 1: The performance of proof logging and verification on the 1,227 benchmark instances. The top left plot shows the cumulative number of instances solved, solved with proof logging, and verified, for increasing time limits. The bottom left plot shows the cumulative number of instances for which the OPB and proof log files are no more than a given size. The top centre scatter plot shows the increase in time required to enable proof logging, whilst the top right scatter plot shows the verification time, in comparison to the time needed to generate proof files; in both cases, lighter point colours indicate larger disk space requirements. The bottom centre scatter plot shows the size of the proof log file, compared to the number of recursive calls made by the solver (and lighter point colours indicate longer verification times). Finally, the bottom right scatter plot shows output sizes as proof logging times increase.

ble of. The bottom right plot of Figure 1 confirms that I/O is our main problem: performance is very closely correlated to the amount of data that is written out.

Secondly, producing the additional constraints for the additional graph pairs can be expensive: although it is a polynomial operation, moving from the solver's $O(|\,\mathrm{V}(P)^2| + |\,\mathrm{V}(T)|^2)$ size requirements to the potential $O(|\,\mathrm{V}(P)^2||\,\mathrm{V}(T)|^2)$ size needed for a PB model can be prohibitive. The additional graph pairs make this worse: they can be either sparser or denser than the inputs, and there are instances where the OPB file is relatively small, but where the additional graph pair constraints are close to the worst possible size. We can see this in the middle column of Figure 1: there are instances where no search is performed, where producing the additional graph pairs takes many hundreds of seconds and several GBytes of proof log space.

5 Conclusion

We have shown, for the first time, that it is possible to carry out proof logging and verification for a sophisticated graph algorithm—and that we can do so without the proof verifier needing to be aware of any graph theory. Although there were limits on input we could consider, this method gave us a practical way of verifying the solutions to over a thousand instances. This is especially helpful because some of these instances were too hard for any other solver, meaning we were not previously completely confident that the Glasgow solver was obtaining correct results through legitimate means.

We hope that solvers for other NP-complete problems will start adopting this technology, particularly since increasingly sophisticated reasoning is now being implemented and used in practice. Although the overheads mean it may not be as practical to use proof logging for all instances as it is in the SAT community, we would still prefer to see solvers which could output proofs at least some of the time. For this reason, we consider it particularly relevant that introducing proof logging into the Glasgow subgraph solver was straightforward and non-intrusive. The combination of RUP and simple justifications for counting arguments meant that our main implementation difficulties came from remembering to handle all the special cases like loops and directed and labelled edges, rather than from proof logging itself. We would be especially interested to see whether cutting planes proofs are similarly effective in other domains.

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 the Engineering and Physical Sciences Research Council [grant number EP/P026842/1]. Some experiments used resources provided by the Swedish National Infrastructure for Computing (SNIC) at the High Performance Computing Center North (HPC2N) at Umeå University.

References

[Archibald *et al.*, 2019] Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2019.

[Audemard *et al.*, 2014] Gilles Audemard, Christophe Lecoutre, Mouny Samy Modeliar, Gilles Goncalves, and Daniel Cosmin Porumbel. Scoring-based neighborhood dominance for the subgraph isomorphism problem. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2014.

[Bonnici *et al.*, 2013] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis E. Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinformatics*, 14(S-7):S13, 2013.

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

[Cordella *et al.*, 2004] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[Elffers *et al.*, 2020] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all differences using pseudo-boolean reasoning. In *Proceedings of AAAI*, 2020.

[Goldberg and Novikov, 2003] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, 2003.

[Haken, 1985] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[Heule *et al.*, 2013a] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 181–188. IEEE, 2013.

[Heule *et al.*, 2013b] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2013.

[Kotthoff *et al.*, 2016] Lars Kotthoff, Ciaran McCreesh, and Christine Solnon. Portfolios of subgraph isomorphism algorithms. In Paola Festa, Meinolf Sellmann, and Joaquin Vanschoren, editors, *Learning and Intelligent Optimization - 10th International Conference, LION 10, Ischia, Italy, May 29 - June 1, 2016, Revised Selected Papers*, volume 10079 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2016.

[Lecoutre *et al.*, 2007] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 131–136, 2007.

[Lee *et al.*, 2016] Jimmy H. M. Lee, Christian Schulte, and Zichen Zhu. Increasing nogoods in restart-based search. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3426–3433. AAAI Press, 2016.

[McCreesh and Prosser, 2015] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 2015.

[McCreesh *et al.*, 2018] Ciaran McCreesh, Patrick Prosser, Christine Solnon, and James Trimble. When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.*, 61:723–759, 2018.

[Solnon, 2010] Christine Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.*, 174(12-13):850–864, 2010.

[Solnon, 2019] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition - 12th IAPR-TC-15 International Workshop, GbRPR 2019, Tours, France, June 19-21, 2019, Proceedings*, volume 11510 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2019.

[Wetzler *et al.*, 2014] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014.

[Zampelli *et al.*, 2010] Stéphane Zampelli, Yves Deville, and Christine Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3):327–353, 2010.

Paper H

Certifying Solvers for Clique and Maximum Common (Connected) Subgraph Problems

Stephan Gocht[1,2], Ross McBride[3], Ciaran McCreesh[3(✉)], Jakob Nordström[1,2], Patrick Prosser[3], and James Trimble[3]

[1] Lund University, Lund, Sweden
stephan.gocht@cs.lth.se
[2] University of Copenhagen, Copenhagen, Denmark
jn@di.ku.dk
[3] University of Glasgow, Glasgow, Scotland
{ciaran.mccreesh,patrick.prosser}@glasgow.ac.uk,
j.trimble.1@research.gla.ac.uk

Abstract. An algorithm is said to be *certifying* if it outputs, together with a solution to the problem it solves, a proof that this solution is correct. We explain how state of the art maximum clique, maximum weighted clique, maximal clique enumeration and maximum common (connected) induced subgraph algorithms can be turned into certifying solvers by using pseudo-Boolean models and cutting planes proofs, and demonstrate that this approach can also handle reductions between problems. The generality of our results suggests that this method is ready for widespread adoption in solvers for combinatorial graph problems.

1 Introduction

McConnell et al. [40] argue that all algorithm implementations should be *certifying*: that is, along with their output, they should produce an easily verified proof that the output is correct. Given the relative frequency of bugs in constraint programming (CP) solvers and in dedicated algorithms for hard combinatorial problems [7,12,25,42], it would be desirable to see certification becoming a social requirement for all new solvers—as has already happened in the Boolean satisfiability community through proof logging formats such as RUP [27], TraceCheck [5], DRAT [29,30,74], LRAT [13] and GRIT [14]. A proof log is a particular kind of certificate which records the steps taken by a solver in such a way that the correctness of each step can easily be checked, given that

The first and fourth authors were funded by the Swedish Research Council (VR) grant 2016-00782. The fourth author was also supported by the Independent Research Fund Denmark (DFF) grant 9040-00389B. The third, fifth and sixth authors were supported by the Engineering and Physical Sciences Research Council [grant numbers EP/P026842/1 and EP/M508056/1]. Some code development used resources provided by the Swedish National Infrastructure for Computing (SNIC).

all previous steps are known to be correct; the intent is that verifying a proof log should be very simple, even if solvers carry out complex reasoning.

Until recently, it was generally assumed that proof logging for more powerful CP-style solvers would require either very complicated (and hard to verify) certificates that must be aware of every kind of propagation performed by every constraint [72], or an exponential slowdown [22]. However, it has recently been shown that reasoning over pseudo-Boolean formulae can compactly express all-different reasoning [19], as well as all of the reasoning carried out by state-of-the-art subgraph isomorphism solvers [26], despite pseudo-Boolean reasoning not knowing anything about Hall sets, matchings, vertices, degrees, or paths.

The general idea behind this proof logging is that a constraint satisfaction problem (or other hard problem) is compiled to a pseudo-Boolean (PB) formula—that is, a 0–1 integer linear program. Then, either a witness of satisfiability is provided, or a proof showing that the PB formula implies $0 \geq 1$ is given. (Optimisation and enumeration problems are also supported.) The proofs of unsatisfiability demonstrated so far have consisted of a mix of "reverse unit propagation" (RUP) steps [19,24] to describe the backtracking search tree produced by the solver, and assistance in deriving any information used by propagators that is not immediately apparent to unit propagation (such as Hall sets and Hall violators). In this work, we report that this approach can be used in a more general way to obtain certifying algorithms (with proofs that can be checked by the VeriPB verifier [19]) for a range of other hard problems:

- We show how a wide variety of maximum clique algorithms from the literature can all be enhanced with proof logging, using very similar proof techniques. We also explain how to adapt this proof logging method to cover the inference used by a state of the art maximum weight clique solver. Finally, we discuss certification for all maximal clique enumeration algorithms.
- We also demonstrate proof logging for a state of the art CP-style maximum common induced subgraph algorithm, including for the connected variant of the problem.
- Finally, we look at a reduction from maximum common induced subgraph to maximum clique, which outperforms CP approaches on certain graph classes. We show that this reduction can be expressed inside the proof log, so we can take a PB model that was generated for the CP encoding, but then provide a proof from a clique algorithm—this is a bit like channelling [8], but for proofs. There are also clique-like algorithms with a propagator to enforce connectedness. Because the reduction can be viewed as a bijection, we can continue to express the connectedness constraint only on the CP encoding (where it is much easier to understand than on the clique encoding), but still validate clique-like proofs.

Our main conclusion is that proof logging using pseudo-Boolean reasoning is general and powerful enough to concisely describe the inference used in a wide range of combinatorial graph algorithms. Although the current implementation does not scale well enough to deal with the largest instances, it can already

be used to provide, for the first time, fully verifiable proofs of correctness for some highly nontrivial medium-sized instances. Also, even if the overhead is currently too high to have proof logging switched on by default in production, it provides an excellent tool for debugging of nontrivial optimisation techniques during solver development. This is because incorrectly implemented steps are likely to lead to incorrect proofs, which can be detected even when the results produced by the solver happen to be correct. We believe this tool is mature enough for widespread adoption, and that requiring all solvers be able to output proofs would be a natural and desirable step to increase the confidence in the correctness of state-of-the-art solvers.

2 Clique Problems

A clique in a graph is a set of vertices, where every vertex in this set is adjacent to every other in this set. The problem of finding a maximum-sized clique in a graph is broadly applicable, and there are many dedicated solvers for the problem (which we will discuss below). However, as McCreesh et al. [42] note, at least some of these solvers are buggy—including the one [35] which was used as a subcomponent by the winner of the 2019 PACE Implementation Challenge [28]. We therefore begin with a worked example, showing how a machine-verifiable proof could be constructed to demonstrate and prove the correctness of a solution for a simple maximum clique instance.

Consider the graph in Fig. 1. To prove that the maximum clique size of this graph is four, we have to show two things: that it has a clique of four vertices, and that there is no larger clique. To do so, we use the VeriPB proof verifier, which takes two files as its input: a pseudo-Boolean model in the standard OPB format [55], and a proof log which provides a verifiable solution to this model. Therefore, our first step is to encode the problem of finding a maximum clique in this graph as a pseudo-Boolean model. We have a 0–1 variable x_i for each vertex i in the graph, an objective which is to maximise the sum of the vertices taken, and for every non-adjacent pair of vertices, a constraint saying they cannot both be taken simultaneously. In OPB format, this looks like:

```
* #variable= 12 #constraint= 41
min: -1 x1 -1 x2 -1 x3 -1 x4 ...and so on... -1 x11 -1 x12 ;
1 ~x3 1 ~x1 >= 1 ;
1 ~x3 1 ~x2 >= 1 ;
1 ~x4 1 ~x1 >= 1 ;
* ...and a further 38 similar lines for the remaining non-edges
```

Here the first line is a special header comment, the second line specifies that the objective is to minimise $\sum_{i=1}^{12} -x_i$ (i.e. to maximise the number of vertices selected, $\sum_{i=1}^{12} x_i$, but OPB supports only minimisation), and subsequent lines specify constraints. An expression like `1 ~x3 1 ~x1 >= 1` corresponds to the linear inequality $1\overline{x}_3 + 1\overline{x}_1 \geq 1$, where the overline means negation, $\overline{x}_i = 1 - x_i$.

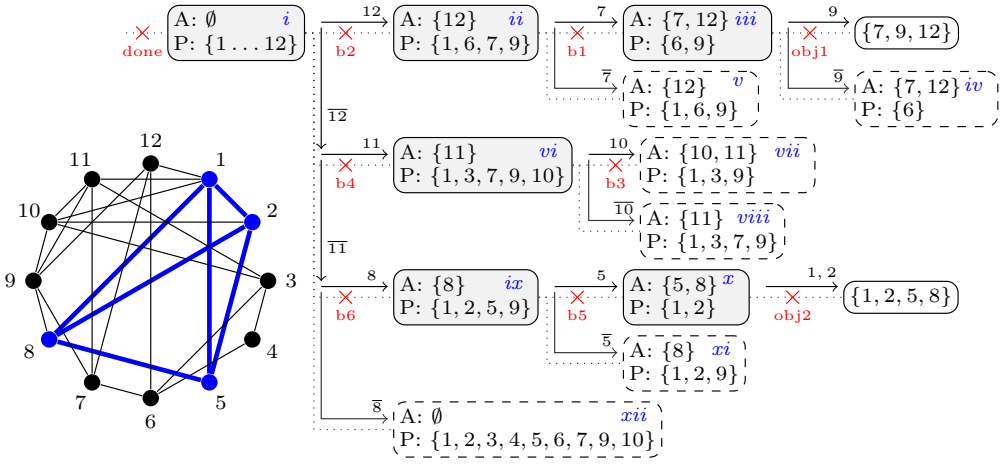


Fig. 1. On the left, a graph, with a 4-vertex clique highlighted. On the right, an illustration of the proof tree used in our worked example to show that this clique is maximum. The solid arrows show the solver's view of the search tree, and are labelled either with a vertex number being accepted or an overlined vertex number being rejected. Shaded boxes represent states in the search tree where we have accepted the vertices labelled "A" and can potentially accept the ones labelled "P", dashed boxes represent states that are eliminated by a bound, and clear boxes are candidate solutions. Roman numerals denote states discussed in the text. Dotted lines show the search tree used by the proof: the crosses with labels correspond to statements that justify a backtrack.

Note the simplicity of this encoding. This is important: the proof we will produce is expressed in terms of this encoding, and because this process is not formally verified, any errors in the encoding could potentially lead to a proof which "proves the wrong thing" being accepted.[1]

Now we move on to the proof. We could produce proofs of the decision problems for 4- and 5-cliques, but the VeriPB format also allows us to verify a branch-and-bound search directly. We now give such a proof, tracing a possible (and intentionally not very good) algorithm execution as we do so. The proof log must begin with a header, as follows (asterisk lines are comments):

```
pseudo-Boolean proof version 1.0
* load the objective function, and the 41 model constraints
f 41 0
```

Typically, maximum clique algorithms maintain two sets during search: a set of accepted vertices, A, which is always a growing clique, and a set of possible vertices P, each of which is adjacent to every vertex in A. Rather than a binary branching scheme, we will iterate over each vertex in P in turn and first accept

[1] We are not aware of any obstacles for providing formal verification for this translation step. However, since this translation is so simple, in this paper we focus on the more challenging task of formally verifying the correctness of solvers' reasoning.

that vertex, then reject it and accept a second vertex instead. We will carry out a typical depth-first branch and bound search, with a rather ad hoc bound for illustration purposes. Our solver will begin in the state labelled i in Fig. 1, with no vertices accepted and every vertex being possible.

Suppose our solver first branches by deciding to accept vertex 12. Then by adjacency, only vertices 1, 6, 7 and 9 remain acceptable; we are in state *ii*. Suppose now we also accept vertex 7. This leaves vertices 6 and 9 possibly to be accepted; we are in state *iii*. We accept vertex 9, which is not adjacent to 6. We have found a maximal clique with three vertices. We therefore record this in the proof log, using an "o" rule. This rule tells the verifier to check that the solution we specified is in fact feasible, and then to create a new constraint, $\sum_{i=1}^{12} x_i \geq 4$, saying that any future solution must be better; this constraint also allows us to backtrack, which is marked as "obj1" in Fig. 1. We log this as:

```
o x7 x9 x12
```

We are now back to having accepted vertices 7 and 12, but now only 6 remains possible; this is state *iv*. Now that we have introduced a new constraint saying we must set at least four variables to true, it is obvious to a human that we are at a dead-end and must backtrack. We now have two options: we can explicitly justify why we can backtrack by deriving a new constraint manually, or we can rely upon some help from the proof verifier.

To derive the constraint manually, we would proceed as follows. If we sum the objective line, every non-adjacency constraint involving x_7 or x_{12}, and the non-adjacency constraint involving x_6 and x_9, we get $\overline{x}_2 + \overline{x}_3 + \overline{x}_4 + \overline{x}_5 + 6\overline{x}_7 + \overline{x}_8 + \overline{x}_{10} + 6\overline{x}_{12} \geq 7$. Now, for any variable x_i, we have an axiom $x_i \geq 0$. By also adding these axioms for each $i \in \{2, 3, 4, 5, 8, 10\}$, and normalising by using the fact that $x_i + \overline{x}_i = 1$, the sum reduces to $6\overline{x}_7 + 6\overline{x}_{12} \geq 1$, which we may then divide by 6 to get $\overline{x}_7 + \overline{x}_{12} \geq 1$ as desired. We *could* express these steps explicitly in the proof log—and we *could* also explain an algorithm a solver could use to know exactly which constraints to sum together and what constant to divide by. But fortunately, there is an easier approach. By using a "u" rule, we may tell the proof verifier to introduce a new constraint which is "obviously" true, given what it knows already. So, we may simply assert:

```
u 1 ~x12 1 ~x7 >= 1 ;
```

and the proof verifier will work out the rest. It is able to do this because this new constraint follows by *reverse unit propagation* (RUP) [19,24]: that is, if we add the negation of this constraint and unit propagate,[2] then contradiction follows without search. We may verify this: the negation of the constraint $\overline{x}_{12} + \overline{x}_7 \geq 1$ is $x_{12} + x_7 \geq 2$. From this, unit propagation infers that both x_7 and x_{12} are 1. Then, using the non-adjacency constraints, all variables except x_6 and x_9

[2] In a PB setting, unit propagation is equivalent to achieving integer bounds consistency [9] on all constraints. This is identical to SAT unit propagation on clausal constraints, but is stronger in general.

will unit propagate to 0. Now, looking at the new objective constraint, we have to set at least four variables to 1, so x_6 and x_9 must both be 1. However, vertices 6 and 9 are non-adjacent, and so there is a constraint forbidding them both to be 1. Thus, RUP can derive a contradiction, and can safely add the asserted constraint—without our hypothetical solver authors having to perform any complicated bookkeeping. This new constraint is labelled "b1" in Fig. 1.

Our solver is now back in the state that it has accepted vertex 12, and it has vertices 1, 6, and 9 to choose from; this is state *v*. Observe that vertices 1 and 6 are non-adjacent, and so it is not possible to make a 4-clique using vertex 12 plus a subset of these vertices. We may therefore backtrack—again, this fact follows using RUP. We label this "b2" in the figure, and log it as:

```
u 1 ~x12 >= 1 ;
```

We are now back at the top of the search tree, having rejected vertex 12 entirely. Suppose we accept vertex 11 next: this leaves vertices 1, 3, 7, 9, and 10 as possibilities, state *vi*. Then suppose we accept vertex 10, leaving vertices 1, 3, and 9 as possibilities, state *vii*. Note that none of these vertices are adjacent, and so we may select at most one of these. To a human, it is now obvious that we may backtrack, but we must give the proof verifier a little help. Before we can use a RUP rule to backtrack, we must derive an at-most-one rule showing that $x_1 + x_3 + x_9 \leq 1$. We may do this as follows:

```
p 1 2 * 19 + 21 + 3 d
p 42 47 +
u 1 ~x11 1 ~x10 >= 1 ;
```

However, these two "p" lines are not easy to read, as expressed: some of the numbers are literal constants, some refer to constraints in the model file, and some refer to constraints we have generated earlier in the verification process. For this discussion, we will therefore take a few liberties with the proof format in our running example. Instead of writing "42" for the objective constraint (which got that number because it was the first introduced constraint, and there are 41 model constraints before it), we will write `obj1`. Similarly, rather than writing 19 to refer to the model constraint $\overline{x}_1 + \overline{x}_9 \geq 1$, we will write `nonadj1_9`. Finally, we will use the notation `⇝ name` to give a name to the result of a rule that we will refer to later on in the proof, or to refer to a point in Fig. 1. After this, any remaining numbers are literal constants. Thus, the above snippet becomes:

```
* at most one [ x1 x3 x9 ]
p nonadj1_3 2 * nonadj1_9 + nonadj3_9 + 3 d                    ⇝ tmp1
p obj1 tmp1 +
u 1 ~x11 1 ~x10 >= 1 ;                                          ⇝ b3
```

and we may explain the two "p" rules more easily. In the cutting planes proof system for pseudo-Boolean formulae [11] upon which VeriPB is based, we can add together existing constraints, multiply existing constraints by a non-negative

integer constant, and divide existing constraints by a positive integer constant. A "p" rule expresses these steps in reverse Polish notation. The first "p" rule multiplies the non-adjacency constraint $\overline{x}_1 + \overline{x}_3 \geq 1$ by 2 to get $2\overline{x}_1 + 2\overline{x}_3 \geq 2$, adds two more non-adjacency constraints to get $3\overline{x}_1 + 3\overline{x}_3 + 2\overline{x}_9 \geq 4$, and divides this by 3 to get the at-most-one constraint $\overline{x}_1 + \overline{x}_3 + \overline{x}_9 \geq 2$. The second "p" rule adds this to our objective constraint, $\sum_{i=1}^{12} x_i \geq 4$, to show that the remaining nine variables must sum to at least 3. This is now sufficient for reverse unit propagation to justify backtracking step "b3".

A very similar argument allows us to backtrack again: having accepted vertex 11, and rejected vertices 10 and 12, we may pick at most one of vertices 1, 3, and 7, plus possibly vertex 9 (state *viii*). We must help the verifier by generating another at-most-one constraint:

```
* at-most-one [ x1 x3 x7 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d                    ⇝ tmp2
p obj1 tmp2 +
u 1 ~x11 >= 1 ;                                                  ⇝ b4
```

We are back at the top of search. Having rejected vertices 11 and 12, if we now branch accepting vertex 8 (state *ix*), and then vertex 5 (state *x*), the remaining possible vertices 1 and 2 can both be added to form a clique. We thus log this as a solution, which generates a new objective constraint $\sum_{i=1}^{12} x_i \geq 5$.

```
o x1 x2 x5 x8                                                    ⇝ obj2
u 1 ~x8 1 ~x5 >= 1 ;                                             ⇝ b5
```

Backtracking to the top of the search tree from state *xi* can be justified by observing that we may pick at most one of vertices 1 and 9:

```
p obj2 nonadj1_9 +
u 1 ~x8 >= 1 ;                                                   ⇝ b6
```

Finally, having rejected vertices 8, 11, and 12 at the top of search, we are in state *xii*, and the remaining nine vertices can be partitioned into independent sets to create three at-most-one constraints. To allow RUP to unset all nine vertices, we will combine these constraints incrementally, as follows.

```
* at-most-one [ x1 x3 x7 ] [ x2 x4 x9 ] [ x5 x6 x10 ]
p nonadj1_3 2 * nonadj1_7 + nonadj3_7 + 3 d                    ⇝ tmp3
p obj2 tmp3 +
p nonadj2_4 2 * nonadj2_9 + nonadj4_9 + 3 d                    ⇝ tmp4
p obj2 tmp3 + tmp4 +
p nonadj5_6 2 * nonadj5_10 + nonadj6_10 + 3 d                  ⇝ tmp5
p obj2 tmp3 + tmp4 + tmp5 +
```

The proof terminates by asserting that we have proved unsatisfiability—that is, there is nothing remaining that can beat the best solution we have found. This is done through a RUP check for contradiction, i.e. that $0 \geq 1$, followed by a "c" rule to terminate the proof.

```
u >= 1 ;                                                    ⇝ done
c done 0
```

Having produced this log, we may now hand it and the associated pseudo-Boolean model file to VeriPB, which will successfully verify it.

There is one other important detail that we have omitted from this proof: in practice, it is extremely helpful to the verifier if we delete temporary constraints when they are used, as well as intermediate backtracking constraints after we have backtracked further up the tree. (This is also crucial for the performance of proof logging for SAT [74].) VeriPB supports both deletion of numbered constraints, and a notion of "levels" which allow all constraints generated below a certain depth to be deleted simultaneously.

2.1 Maximum Clique Algorithms in General

The majority of maximum clique algorithms that are aimed at hard, dense graphs make use of backtracking search with branch and bound [4,33,35–37,39,44,51,53,57,58,60,62,66–69,71]. The inference on adjacency performed by all of these algorithms is straightforward, with all of the cleverness being in branching and how bounds are computed [1]. We may therefore produce proof logs for all of these algorithms using only RUP, logging of solutions as they are found, and some additional help for the bounds.

Colour Bounds. If a graph can be coloured using k colours (where adjacent vertices must be given different colours) then it cannot contain a clique of more than k vertices. Producing an optimal colouring is hard (and typically harder in practice than finding a maximum clique), but various greedy methods exist, and have been used to give a dynamic bound during search inside clique algorithms. Suppose we have, after branching, our set of accepted vertices A, a set of undecided vertices P, and have already found a clique of n vertices. If $c(P)$ is the number of colours used in some legal colouring of the subgraph induced by P, then if $|A| + c(P) \leq n$, we can immediately backtrack.

Using cutting planes, if we are given a colouring then it is easy to produce a proof that this bound is valid. By definition, for each pair of vertices in a given colour class, the PB model must have a constraint saying that both vertices cannot be taken simultaneously (because they do not have an edge between them). As we saw in the worked example, it is routine to combine these constraints into an at-most-one constraint, using a single sequence of arithmetic operations that mentions each pairwise constraint only once. We can then sum these new at-most-one constraints, add them to the objective constraint, and the rest of the work follows by unit propagation.

Incremental Colour Bounds. Producing a colouring can be relatively expensive. In order to reduce the number of colourings needed, many solvers reuse colourings. Suppose we have produced colour classes $C_1, \ldots, C_c$. Instead of making a single branching decision, we may branch on accepting each vertex in colour class

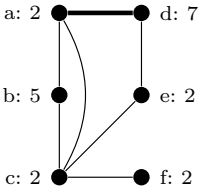
C_c in turn first, followed by those in C_{c-1}, then C_{c-2} and so on, stopping after we have visited only $n - |A| + 1$ colour classes. Ideally, in a proof log, we would not have to produce individual statements to justify not exploring each vertex in each remaining colour class. This is indeed possible: we derive an at-most-one constraint for colour class C_1, and remember its number ℓ_1. We then add this constraint to the objective constraint. Next, we derive an at-most-one constraint for colour class C_2, add this to ℓ_1, and remember its number ℓ_2. Now we sum the objective constraint, ℓ_1, and ℓ_2. We continue until we reach a colour class which was used for branching—again, the worked example made use of this.

Other changes to the details of how colour bounds are produced has formed a substantial line of work in maximum clique algorithms [33,51,53,57,58,62,66–69]. However, proof logging is completely agnostic to this: we care only that we have a valid colouring, and do not need to understand any of the details of the algorithm that produced it.

Stronger Bounds. Even when a good colouring is found, colour bounds can be quite weak in practice. Some clique solvers identify subsets of k colour classes which cannot form a clique of k vertices. For example, San Segundo et al. [60] will find certain cases where there is a pair of colour classes C_1 and C_2, together with a vertex v, such that no triangle exists using v and a vertex each from C_1 and C_2, and uses this to reduce the bound by one for vertex v. If such a case is identified, RUP is sufficient to justify it. Similarly, because pseudo-Boolean unit propagation is at least as strong as SAT unit propagation, bounds using MaxSAT reasoning on top of colour classes [35–37] are also easily justified.

Algorithm Features Not Affecting Proof Logging. Maximum clique algorithms have used a variety of different search orders [44]; as with the details of how colourings are produced, these details are irrelevant for proof logging. Similarly, bit-parallelism [59,61] has no effect on proof logging; thread-parallelism [16,43,45] remains to be seen, but since proof logging is largely I/O bound, it is likely that gains from multi-core parallelism will be lost on current hardware when logging. And finally, running a local search algorithm and "priming" the branch and bound algorithm with a strong initial incumbent [4,39,71] requires only that the new incumbent be logged before the search starts, regardless of how that incumbent was found.

Implementation. We implemented proof logging for the dedicated clique solver which is included in the Glasgow Subgraph Solver [48], and tested it on a system with dual Intel Xeon E5-2697A v4 CPUs, 512 GBytes RAM, and a conventional hard drive, running Ubuntu 18.04. Without proof logging, this solver is able to solve 59 of the 80 instances from the second DIMACS implementation challenge [32] in under 1,000 s. With proof logging enabled, we produced proof logs for 57 of the 59 instances, incurring a mean slowdown of 80.1; the final two instances were cancelled when their proof logs reached 1 TByte in size. We were then able to verify all 57 of these proofs, with verification being a mean of 10.1 times more expensive than writing the proofs. Note that the logging slowdown is to




```
pseudo-Boolean proof version 1.0
f 8 0
o xa xd                                                    ⇝ obj
p nonadja_e 2 * nonadja_f + nonadje_f + 3 d 2 *            ⇝ cc1
p nonadjb_d 5 *                                            ⇝ cc2
p nonadjc_d 2 *                                            ⇝ cc3
p obj cc1 + cc2 + cc3 +                                    ⇝ done
c done 0
```

Fig. 2. On the left, a weighted graph, with a clique of weight ten from vertices a and d highlighted. On the right, a proof that there is no heavier clique.

be expected [26]: the original solver is able to carry out a full recursive call and bounds calculation in under a microsecond. If each such call requires 1 KByte of logged information then this already exceeds the 100 MBytes per second write capabilities of a hard disk by an order of magnitude.

2.2 Weighted Clique Algorithms

In the maximum weight clique problem, vertices have weights, and we are now looking to find the clique with the largest sum of the weights of its vertices, rather than the most vertices. A simple bound for this problem is to produce a colouring, and then sum the maximum weight of each colour class. Consider the example in Fig. 2, and the three colour classes $\{a, e, f\}$, $\{b, d\}$ and $\{c\}$. By looking only at the largest weight in each colour class, we obtain a bound of $2+7+2=11$. This bound may be justified in a cutting planes proof by generating the at-most-one constraints for each colour class as previously, and then multiplying each colour class by its maximum weight before summing them. However, better bounds can be produced by allowing a vertex to appear in multiple colour classes, and by splitting its weight among these colour classes. If we allow vertex d to appear in the second colour class with weight 5 and in the third colour class with weight 2, then our bound is $2+5+2=9$. This technique originates with Babel [2], and is used in algorithms due to Tavares et al. [64,65], which are the current state of the art for many graph classes [46]. From a proof logging perspective, this splitting does not affect how we generate the bound, and so we may generate the proof shown on the right of Fig. 2.

Implementation. We implemented a simple certifying maximum weight clique algorithm using the Tavares et al. [64,65] bound in Python. With a timeout of 1,000 s, we were able to produce proof logs for 174 of the 289 benchmark instances from a recent collection [46]; all were verified successfully.

2.3 Maximal Clique Enumeration

Finally, in some applications we want to find every maximal clique (that is, one which cannot be made larger by adding vertices without removing vertices).

This problem also has a straightforward PB encoding: we express maximality by having a constraint for every vertex v saying that either x_v is selected, or at least one of its non-neighbours is.

The classic Bron-Kerbosch algorithm [6] uses a simple backtracking search, employing special data structures to minimise memory usage; it ensures maximality through a data structure called a *not*-set. We do not explain this data structure here, because it turns out to be equivalent in strength to unit propagation on the above PB model—indeed, to create a proof-logging Bron-Kerbosch algorithm, one needs only output a statement for every found solution, and a statement on every backtrack. More recent variations on this algorithm make use of different branching techniques [20,49,56,70] and supporting data structures [15,21,56], but although these new techniques can make a huge difference to theoretical worst-case guarantees and to empirical runtimes, they do not require any changes to how proof logging is performed.

We are interested in proof logging for this problem because there are discrepancies in tables of published results for some common benchmark instances—for example, does the "celegensneural" instance from the Newman dataset have 856 [21], 1,386 [20], or some other number of maximal cliques? We implemented proof logging for Tomita et al.'s variant of the algorithm [70], and were able to confirm that 1,386 is the correct answer. We were also able to confirm the published values of Eppstein et al. [20] for all of the BioGRID instances, the listed DIMACS instances, and for the Newman instances with no more than 10,000 vertices. We were unable to produce certified results for larger sparse graphs, because the OPB encoding size is linear in the number of *non*-edges in the inputs.

3 Maximum Common Induced Subgraph Algorithms

The maximum common induced subgraph problem can be defined in various equivalent ways, but the most useful is that we are given two graphs, and must find an injective partial mapping from the first graph to the second, where adjacent vertices are mapped to adjacent vertices and non-adjacent vertices are mapped to non-adjacent vertices, mapping as many vertices as possible. The problem arises in applications including in chemistry and biology [18,23,54]. However, in many cases, the common subgraph is required to be *connected*: that is, if we take any two assigned vertices from the first graph, then we must be able to find a path from one to the other without using unassigned vertices.

McCreesh et al. [41] compared two approaches to the problem, one based upon CP [50,73] and one based upon reduction to clique [3,17,34,54], and found that the best approach varied depending upon the kinds of graph being used. Since then, improvements have come from two different lines of research: one based upon weakening subgraph isomorphism algorithms [31], and one called McSplit which replaces general algorithms and data structures used in CP with much faster domain-specific ones [38,47]. We will discuss CP and McSplit, and then the clique reduction later, but first we must provide an appropriate PB encoding.

3.1 Pseudo-Boolean Encodings

To encode maximum common induced subgraph in PB form, we may adapt the subgraph isomorphism encoding of Gocht et al. [26]. For each vertex f in the first graph F, and for each vertex s in the second graph S, we have a variable $x_{f,s}$ which takes the value 1 if f is mapped to s; we also have a variable $x_{f,\perp}$ if f is unassigned. We then have exactly-one constraints over each set of $x_{f,-}$ variables, at-most-one constraints over each set of $x_{-,s}$ variables for injectivity, and induced adjacency constraints which are expressed using Gocht et al.'s second encoding,

$$
\begin{aligned}
x_{f,\perp} + \sum_{s \in \mathrm{V}(S)} x_{f,s} = 1 \qquad & f \in \mathrm{V}(F) \\
\sum_{f \in \mathrm{V}(F)} x_{f,s} \leq 1 \qquad & s \in \mathrm{V}(S) \\
\overline{x}_{f,s} + x_{g,\perp} + \sum_{t \in \mathrm{N}(s)} x_{q,t} \geq 1 \qquad & f \in \mathrm{V}(F),\ q \in \mathrm{N}(f),\ s \in \mathrm{V}(S) \\
\overline{x}_{f,s} + x_{g,\perp} + \sum_{t \in \overline{\mathrm{N}}(s)} x_{q,t} \geq 1 \qquad & f \in \mathrm{V}(F),\ q \in \overline{\mathrm{N}}(f),\ s \in \mathrm{V}(S)
\end{aligned}
$$

and the objective is to maximise the sum of the non-$\perp$ variables.

For the connected version of the problem, expressing connectedness as a constraint is a little trickier. Our encoding is informed by two simple observations: a subgraph with k vertices is connected if, for every pair of vertices in the subgraph, there is a walk of length no more than k between them, and secondly, for $k > 1$, there is a walk of length $2k$ between two vertices f and g if and only if there is some vertex h such that there are walks of length k between f and h and also between h and g.

Therefore, we first introduce auxiliary variables $x^1_{f,g}$ for every pair of vertices f and g in the first graph.[3] If f and g are non-adjacent, these variables are forced to false; otherwise we add constraints to specify that $x^1_{f,g}$ is true if and only if both $x_{f,\perp}$ and $x_{g,\perp}$ are false. In other words, $x^1_{f,g}$ is true precisely if f and g are adjacent and in the chosen subgraph. Writing $f \sim_F g$ and $f \not\sim_F g$ to mean vertices f and g are adjacent or not adjacent in the graph F respectively, this is:

$$
\begin{aligned}
\overline{x}^1_{f,g} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \not\sim_F g \\
\overline{x}^1_{f,g} + \overline{x}_{f,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g \\
\overline{x}^1_{f,g} + \overline{x}_{g,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g \\
x^1_{f,g} + x_{f,\perp} + x_{g,\perp} \geq 1 \qquad & f, g \in \mathrm{V}(F), f \sim_F g
\end{aligned}
$$

Next, we introduce auxiliary variables $x^2_{f,g}$, which will tell us if there is a walk of length 2 between vertices f and g. To do this, for each other vertex h,

[3] In all of what follows, these variables are equivalent under the exchange of f and g, and so we may halve the number of variables needed by exchanging f and g if $f > g$. We do this in practice, but omit this in the description for clarity.

we have a variable $x^2_{f,h,g}$ which we constrain to be true if and only if there is a walk of length 1 from f to h, and from h to g. Now, $x^2_{f,g}$ may be constrained to be true if and only if either there is a walk of length 1 between f and g, or at least one $x^2_{f,h,g}$ variable is true. We then repeat this process for walks of length 4, 8, and so on, until we reach a length k which equals or exceeded the number of vertices in the first graph. For $k \in \{2, 4, 8, \dots, 2^{\lceil \log |\mathrm{V}(F)| \rceil}\}$:

$$
\begin{aligned}
x^{k/2}_{f,h} + \overline{x}^k_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^{k/2}_{h,g} + \overline{x}^k_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^k_{f,h,g} + \overline{x}^{k/2}_{f,h} + \overline{x}^{k/2}_{h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
\overline{x}^k_{f,g} + x^{k/2}_{f,g} + \sum_{h \in \mathrm{V}(F) \setminus \{f,g\}} x^k_{f,h,g} &\geq 1 && f, g \in \mathrm{V}(F) \\
x^k_{f,g} + \overline{x}^k_{f,h,g} &\geq 1 && f, g, h \in \mathrm{V}(F), h \neq f, h \neq g \\
x^k_{f,g} + \overline{x}^{k/2}_{f,g} &\geq 1 && f, g \in \mathrm{V}(F)
\end{aligned}
$$

Finally, to enforce connectedness, for each pair of vertices f and g, we require that either $x_{f,\perp}$ or $x_{g,\perp}$ or $x^k_{f,g}$ is true.

$$
x_{f,\perp} + x_{g,\perp} + x^k_{f,g} \geq 1 \qquad\qquad f, g \in \mathrm{V}(F), k = 2^{\lceil \log |\mathrm{V}(F)| \rceil}
$$

An important property of this encoding is that all the auxiliary variables are *dependent*: that is, for every solution to the maximum common connected induced subgraph problem, there is exactly one feasible way of setting the auxiliary variables. In other words, the number of solutions to the PB encoding is exactly the same as the number of solutions to the real problem.

3.2 Proof Logging for Constraint Programming Algorithms

The McSplit algorithm [47] performs a CP-style backtracking search [50,73], looking to map as many vertices from the first graph as possible to distinct vertices in the second graph. We will therefore continue to use RUP to generate proofs. For adjacency and non-adjacency constraints, McSplit's reasoning is equivalent to unit propagation on our PB constraints, and so no help is needed. For the bound, McSplit performs "all different except $\perp$" propagation, but with the number of occurrences of $\perp$ constrained to beat the best solution found so far. Due to the special structure of the domains during search, it is able to do this in linear time, without needing the usual matching and components algorithm [52]. However, when it fails, it produces a sequence of Hall sets, and so we may reuse the justification technique described by Elffers et al. [19] with only a simple modification to cope with the objective function.

For the connected variant, McSplit uses a restricted branching scheme [47,73] rather than a conventional propagator: once at least one vertex is assigned a

non-null value, it may only branch on vertices adjacent to a vertex already assigned a non-null value. If no such vertices exist, it backtracks. Interestingly, this requires no explicit support in proof logging: by carefully stepping through the auxiliary variables in the PB encoding, level by level, it can be seen that RUP will propagate all remaining variables to false when in this situation.

Therefore, implementing proof logging in McSplit requires four kinds of statement to be logged. Firstly, any new incumbent must be noted, as in the previous section. Secondly, all backtracks must be logged using a RUP rule. Thirdly, whenever the bound function detects that the current state may be pruned, we must derive a new constraint justifying this. And fourthly, it is extremely helpful to delete intermediate constraints using "level" statements. Again, this proof logging is completely agnostic to changes to the branching heuristic [38].

We implemented this proof logging inside the original McSplit implementation, and tested it for both the connected and non-connected variants of the problem on a commonly used set of benchmark instances [10,63]. We successfully verified McSplit's solutions to all 16,300 instances of no more than 25 vertices. Proof logging introduced a mean slowdown of 67.0 and 298.9 for non-connected and connected respectively, whilst verification was a further 13.4 and 21.6 times slower; again, writing to hard disk was by far the biggest bottleneck, as McSplit can make over five million recursive calls per second.

3.3 Maximum Common (Connected) Subgraph via Clique

An alternative approach to the maximum common subgraph problem is via a reduction to the maximum clique problem [3,34,54]. This reduction resembles the microstructure encoding of the CP representation, and is the best known approach on labelled graphs; we refer to McCreesh et al. [41] for a detailed explanation. From a proof logging perspective, one might expect that this encoding would require a whole new PB representation, or perhaps a large change to how proof logging is performed by a maximum clique algorithm. However, this is not the case: given the PB model for a maximum common subgraph problem from earlier in this section, we can derive the non-adjacency constraints needed for the clique model described in the previous section using only RUP, whilst the objective function needs no rewriting at all. Therefore, the only changes needed to a proof-logging clique algorithm is in the lookup of constraint identifiers.

McCreesh et al. [41] also show how a maximum clique algorithm can be adapted to deal with the connected variant of the problem, by embedding a propagator inside the clique algorithm. From a proofs perspective, we can work with the PB model and the clique reformulation, similar to channelling [8]—and since connectedness propagation requires no explicit proof logging with the original PB representation, it also requires no proof logging when performed inside a clique algorithm.

We therefore reimplemented McCreesh et al.'s clique common (connected) subgraph algorithm [41], and added proof logging support. Proof logging immediately caught a bug in our reimplementation that testing had failed to identify: we were only updating the incumbent when a maximal clique was found, which

is correct in conventional clique algorithms but not for the connected variant, but this very rarely caused incorrect results. Once corrected, for both variants of the problem, we were able to verify all 11,400 instances of no more than 20 vertices from the same set of instances [10,63]. Proof logging introduced an average slowdown of 28.6 and 39.7 for non-connected and connected respectively, and verification was on average a further 11.3 and 73.1 times slower.

4 Conclusion

We have shown that pseudo-Boolean proof logging is sufficiently powerful and flexible to make certification possible for a wide range of graph solvers. Particularly of note is how proof logging is largely agnostic towards most changes to details in algorithm behaviour (such as search order, methods for calculating bounds, and underlying algorithms and data structures), and how it is able to deal with reformulation or changes of representation. This suggests that requiring certification should not be an undue burden on solver authors going forward. We also stress the simplicity of implementation: for every algorithm we considered, proof logging only required access to information that was already easily available inside the existing solvers. In particular, we do not need to implement any form of pseudo-Boolean constraint processing in order to generate these proofs, nor does a solver have to in any way understand or otherwise reason about the proofs it is producing. Furthermore, in each case, adding in support for proof logging was considerably easier than implementing the algorithm itself.

It is important to remember that proof logging does not prove that any algorithm or solver is correct. Instead, it provides a proof that a *claimed solution* is correct—and if a solution was produced using unsound reasoning, this will be caught, even if the solution is correct, or if it was produced by a correct algorithm being run on faulty hardware or with a buggy compiler. Additionally, this process does not verify that the encoding from a high level model to the PB representation is correct. To offset this, the encodings we use are deliberately simple, and when a more complex internal representation is used (such as in the clique model for maximum common subgraph), we can log the reformulation and verify the log in terms of the simpler model. This reformulation also suggests that for competitions, providing a standard encoding would not be a problem.

Although proof logging introduces considerable overheads (particularly when compared to the techniques used in the SAT community, which do not need to deal with powerful but highly efficient propagators), it can still be used to verify medium-sized instances involving tens of millions of inference steps. Given the abundance of buggy solver implementations that *usually* produce correct answers, we suggest that all authors of dedicated graph solvers should adopt proof logging from now on, and that competition organisers should strongly consider requiring proof logging support from entrants. For larger and harder instances, proof logging can be disabled, but because proof logging does not require intrusive changes to solver internals, this would still give us a large increase in confidence in the correctness of results compared to conventional testing.

References

1. Atserias, A., Bonacina, I., de Rezende, S.F., Lauria, M., Nordström, J., Razborov, A.A.: Clique is hard on average for regular resolution. In: Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, 25–29 June 2018, pp. 866–877 (2018)
2. Babel, L.: A fast algorithm for the maximum weight clique problem. Computing **52**(1), 31–38 (1994)
3. Balas, E., Yu, C.S.: Finding a maximum clique in an arbitrary graph. SIAM J. Comput. **15**(4), 1054–1068 (1986)
4. Batsyn, M., Goldengorin, B., Maslov, E., Pardalos, P.M.: Improvements to MCS algorithm for the maximum clique problem. J. Comb. Optim. **27**(2), 397–416 (2014)
5. Biere, A.: Tracecheck (2006). http://fmv.jku.at/tracecheck/
6. Bron, C., Kerbosch, J.: Finding all cliques of an undirected graph (algorithm 457). Commun. ACM **16**(9), 575–576 (1973)
7. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_6
8. Cheng, B.M.W., Lee, J.H.M., Wu, J.C.K.: Speeding up constraint propagation by redundant modeling. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 91–103. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61551-2_68
9. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Sattar, A., Kang, B. (eds.) AI 2006. LNCS (LNAI), vol. 4304, pp. 49–58. Springer, Heidelberg (2006). https://doi.org/10.1007/11941439_9
10. Conte, D., Foggia, P., Vento, M.: Challenging complexity of maximum common subgraph detection algorithms: a performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. **11**(1), 99–143 (2007)
11. Cook, W.J., Coullard, C.R., Turán, G.: On the complexity of cutting-plane proofs. Discret. Appl. Math. **18**(1), 25–38 (1987)
12. Cook, W.J., Koch, T., Steffy, D.E., Wolter, K.: A hybrid branch-and-bound approach for exact rational mixed-integer programming. Math. Program. Comput. **5**(3), 305–344 (2013)
13. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 220–236. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_14
14. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 118–135. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_7
15. Dasari, N.S., Ranjan, D., Zubair, M.: pbitMCE: a bit-based approach for maximal clique enumeration on multicore processors. In: 20th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2014, Hsinchu, Taiwan, 16–19 December 2014, pp. 478–485 (2014)
16. Depolli, M., Konc, J., Rozman, K., Trobec, R., Janezic, D.: Exact parallel maximum clique algorithm for general and protein graphs. J. Chem. Inf. Model. **53**(9), 2217–2228 (2013)
17. Durand, P.J., Pasari, R., Baker, J.W., Tsai, C.C.: An efficient algorithm for similarity analysis of molecules. Internet J. Chem. **2**(17), 1–16 (1999)

18. Ehrlich, H.C., Rarey, M.: Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. Wiley Interdiscip. Rev.: Comput. Mol. Sci. **1**(1), 68–79 (2011)
19. Elffers, J., Gocht, S., McCreesh, C., Nordström, J.: Justifying all differences using pseudo-boolean reasoning. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, 7–12 February 2020, pp. 1486–1494 (2020)
20. Eppstein, D., Löffler, M., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. ACM J. Exp. Algorithmics **18** (2013)
21. Eppstein, D., Strash, D.: Listing all maximal cliques in large sparse real-world graphs. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 364–375. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20662-7_31
22. Gange, G., Stuckey, P.J.: Certifying optimality in constraint programming, February 2019. Presentation at KTH Royal Institute of Technology. Slides https://www.kth.se/polopoly_fs/1.879851.1550484700!/CertifiedCP.pdf
23. Gay, S., Fages, F., Martinez, T., Soliman, S., Solnon, C.: On the subgraph epimorphism problem. Discret. Appl. Math. **162**, 214–228 (2014)
24. Gelder, A.V.: Verifying RUP proofs of propositional unsatisfiability. In: International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, 2–4 January 2008 (2008)
25. Gillard, X., Schaus, P., Deville, Y.: SolverCheck: declarative testing of constraints. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 565–582. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_33
26. Gocht, S., McCreesh, C., Nordström, J.: Subgraph isomorphism meets cutting planes: solving with certified solutions. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020 [scheduled for July 2020, Yokohama, Japan, postponed due to the Corona pandemic], pp. 1134–1140 (2020)
27. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), Munich, Germany, 3–7 March 2003, pp. 10886–10891. IEEE Computer Society (2003)
28. Hespe, D., Lamm, S., Schulz, C., Strash, D.: WeGotYouCovered: the winning solver from the PACE 2019 implementation challenge, vertex cover track. CoRR abs/1908.06795 (2019)
29. Heule, M., Hunt Jr., W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 181–188 (2013)
30. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Verifying refutations with extended resolution. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 345–359. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_24
31. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, San Francisco, California, USA, 4–9 February 2017, pp. 3907–3914 (2017)

32. Johnson, D.S., Trick, M.A.: Introduction to the second DIMACS challenge: cliques, coloring, and satisfiability. In: Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, 11–13 October 1993, pp. 1–10 (1993)
33. Konc, J., Janežič, D.: An improved branch and bound algorithm for the maximum clique problem. MATCH Commun. Math. Comput. Chem. **58**(3), 569–590 (2007)
34. Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. CALCOLO **9**(4), 341–352 (1973). https://doi.org/10.1007/BF02575586
35. Li, C., Jiang, H., Manyà, F.: On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. Comput. Oper. Res. **84**, 1–15 (2017)
36. Li, C.-M., Jiang, H., Xu, R.-C.: Incremental MaxSAT reasoning to reduce branches in a branch-and-bound algorithm for MaxClique. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 268–274. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19084-6_26
37. Li, C.M., Quan, Z.: An efficient branch-and-bound algorithm based on MaxSAT for the maximum clique problem. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010 (2010)
38. Liu, Y., Li, C., Jiang, H., He, K.: A learning based branch and bound for maximum common subgraph related problems. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, 7–12 February 2020, pp. 2392–2399 (2020)
39. Maslov, E., Batsyn, M., Pardalos, P.M.: Speeding up branch and bound algorithms for solving the maximum clique problem. J. Glob. Optim. **59**(1), 1–21 (2014)
40. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Comput. Sci. Rev. **5**(2), 119–161 (2011)
41. McCreesh, C., Ndiaye, S.N., Prosser, P., Solnon, C.: Clique and constraint models for maximum common (connected) subgraph problems. In: Rueher, M. (ed.) CP 2016. LNCS, vol. 9892, pp. 350–368. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44953-1_23
42. McCreesh, C., Pettersson, W., Prosser, P.: Understanding the empirical hardness of random optimisation problems. In: Schiex, T., de Givry, S. (eds.) CP 2019. LNCS, vol. 11802, pp. 333–349. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30048-7_20
43. McCreesh, C., Prosser, P.: Multi-threading a state-of-the-art maximum clique algorithm. Algorithms **6**(4), 618–635 (2013)
44. McCreesh, C., Prosser, P.: Reducing the branching in a branch and bound algorithm for the maximum clique problem. In: O'Sullivan, B. (ed.) CP 2014. LNCS, vol. 8656, pp. 549–563. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10428-7_40
45. McCreesh, C., Prosser, P.: The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. ACM Trans. Parallel Comput. **2**(1), 8:1–8:27 (2015)
46. McCreesh, C., Prosser, P., Simpson, K., Trimble, J.: On maximum weight clique algorithms, and how they are evaluated. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 206–225. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66158-2_14

47. McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, 19–25 August 2017, pp. 712–719 (2017)
48. McCreesh, C., Prosser, P., Trimble, J.: The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants. In: Gadducci, F., Kehrer, T. (eds.) ICGT 2020. LNCS, vol. 12150, pp. 316–324. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51372-6_19
49. Naudé, K.A.: Refined pivot selection for maximal clique enumeration in graphs. Theor. Comput. Sci. **613**, 28–37 (2016)
50. Ndiaye, S.N., Solnon, C.: CP models for maximum common subgraph problems. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 637–644. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_48
51. Nikolaev, A., Batsyn, M., Segundo, P.S.: Reusing the same coloring in the child nodes of the search tree for the maximum clique problem. In: Dhaenens, C., Jourdan, L., Marmion, M.-E. (eds.) LION 2015. LNCS, vol. 8994, pp. 275–280. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19084-6_27
52. Petit, T., Régin, J.-C., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 451–463. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45578-7_31
53. Prosser, P.: Exact algorithms for maximum clique: a computational study. Algorithms **5**(4), 545–587 (2012)
54. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. J. Comput. Aided Mol. Des. **16**(7), 521–533 (2002)
55. Roussel, O., Manquinho, V.M.: Input/output format and solver requirements for the competitions of pseudo-Boolean solvers, January 2016. Revision 2324. http://www.cril.univ-artois.fr/PB16/format.pdf
56. San Segundo, P., Furini, F., Artieda, J.: A new branch-and-bound algorithm for the maximum weighted clique problem. Comput. Oper. Res. **110**, 18–33 (2019)
57. San Segundo, P., Lopez, A., Batsyn, M.: Initial sorting of vertices in the maximum clique problem reviewed. In: Pardalos, P.M., Resende, M.G.C., Vogiatzis, C., Walteros, J.L. (eds.) LION 2014. LNCS, vol. 8426, pp. 111–120. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09584-4_12
58. San Segundo, P., Lopez, A., Batsyn, M., Nikolaev, A., Pardalos, P.M.: Improved initial vertex ordering for exact maximum clique search. Appl. Intell. **45**(3), 868–880 (2016)
59. San Segundo, P., Matía, F., Rodríguez-Losada, D., Hernando, M.: An improved bit parallel exact maximum clique algorithm. Optim. Lett. **7**(3), 467–479 (2013)
60. San Segundo, P., Nikolaev, A., Batsyn, M., Pardalos, P.M.: Improved infra-chromatic bound for exact maximum clique search. Informatica Lith. Acad. Sci. **27**(2), 463–487 (2016)
61. San Segundo, P., Rodríguez-Losada, D., Jiménez, A.: An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Res. **38**(2), 571–581 (2011)
62. San Segundo, P., Tapia, C.: Relaxed approximate coloring in exact maximum clique search. Comput. Oper. Res. **44**, 185–192 (2014)
63. Santo, M.D., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recogn. Lett. **24**(8), 1067–1079 (2003)
64. Tavares, A.W.: Algoritmos exatos para problema da clique maxima ponderada. Ph.D. thesis, Universidade federal do Ceará (2016)

65. Tavares, W.A., Neto, M.B.C., Rodrigues, C.D., Michelon, P.: Um algoritmo de branch and bound para o problema da clique máxima ponderada. In: Proceedings of XLVII SBPO, vol. 1 (2015)
66. Tomita, E., Kameda, T.: An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. J. Glob. Optim. **37**(1), 95–111 (2007)
67. Tomita, E., Seki, T.: An efficient branch-and-bound algorithm for finding a maximum clique. In: Calude, C.S., Dinneen, M.J., Vajnovszki, V. (eds.) DMTCS 2003. LNCS, vol. 2731, pp. 278–289. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45066-1_22
68. Tomita, E., Sutani, Y., Higashi, T., Takahashi, S., Wakatsuki, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique. In: Rahman, M.S., Fujita, S. (eds.) WALCOM 2010. LNCS, vol. 5942, pp. 191–203. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11440-3_18
69. Tomita, E., Sutani, Y., Higashi, T., Wakatsuki, M.: A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments. IEICE Trans. Inf. Syst. **96-D**(6), 1286–1298 (2013)
70. Tomita, E., Tanaka, A., Takahashi, H.: The worst-case time complexity for generating all maximal cliques and computational experiments. Theor. Comput. Sci. **363**(1), 28–42 (2006)
71. Tomita, E., Yoshida, K., Hatta, T., Nagao, A., Ito, H., Wakatsuki, M.: A much faster branch-and-bound algorithm for finding a maximum clique. In: Zhu, D., Bereg, S. (eds.) FAW 2016. LNCS, vol. 9711, pp. 215–226. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39817-4_21
72. Veksler, M., Strichman, O.: A proof-producing CSP solver. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010 (2010)
73. Vismara, P., Valery, B.: Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In: Le Thi, H.A., Bouvry, P., Pham Dinh, T. (eds.) MCO 2008. CCIS, vol. 14, pp. 358–368. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87477-5_39
74. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31

Paper I 

On Division Versus Saturation in Pseudo-Boolean Solving

Stephan Gocht[1] , **Jakob Nordström**[2,1] and **Amir Yehudayoff**[3]
[1]KTH Royal Institute of Technology
[2]University of Copenhagen
[3]Technion – Israel Institute of Technology
gocht@kth.se, jn@di.ku.dk, amir.yehudayoff@gmail.com

Abstract

The conflict-driven clause learning (CDCL) paradigm has revolutionized SAT solving over the last two decades. Extending this approach to pseudo-Boolean (PB) solvers doing 0-1 linear programming holds the promise of further exponential improvements in theory, but intriguingly such gains have not materialized in practice. Also intriguingly, most PB extensions of CDCL use not the division rule in cutting planes as defined in [Cook *et al.*, '87] but instead the so-called saturation rule. To the best of our knowledge, there has been no study comparing the strengths of division and saturation in the context of conflict-driven PB learning, when all linear combinations of inequalities are required to cancel variables.

We show that PB solvers with division instead of saturation can be exponentially stronger. In the other direction, we prove that simulating a single saturation step can require an exponential number of divisions. We also perform some experiments to see whether these phenomena can be observed in actual solvers. Our conclusion is that a careful combination of division and saturation seems to be crucial to harness more of the power of cutting planes.

1 Introduction

Although the Boolean satisfiability (SAT) problem is NP-complete [Cook, 1971], and hence expected to be intractable from a theoretical point of view, there has been enormous progress in performance in the last 15–20 years of SAT solvers based on *conflict-driven clause learning (CDCL)* [Marques-Silva and Sakallah, 1999; Moskewicz *et al.*, 2001].[1] Today CDCL solvers are routinely used for large-scale real-world problems in a wide range of areas [Biere *et al.*, 2009].

Annoyingly, however, there also exist tiny formulas that are completely beyond reach even for the best CDCL solvers, which highlights two limitations of this approach:

- The conjunctive normal form (CNF) used for CDCL input is a weak formalism for encoding constraints.
- The *resolution* method of reasoning used in CDCL solvers is quite weak.

Pseudo-Boolean (PB) constraints can be exponentially more concise than CNF, and PB reasoning (which can be thought of as 0-1 integer linear programming with conflict analysis) is exponentially more powerful than resolution in theory. Extending the conflict-driven framework to a pseudo-Boolean setting would therefore seem like an attractive option. However, although there are crafted benchmark formulas on which PB solvers exponentially outperform CDCL-based approaches, in practice they are often less efficient.

In this work, we study the rules of reasoning used in PB solvers and how they compare to the cutting planes method on which they are based. Interestingly, the most popular conflict-driven PB solvers use the so-called saturation rule instead of the division rule in [Cook *et al.*, 1987]. Our focus is on understanding the relative strengths of division and saturation.

Let us review some background. In this paper, by pseudo-Boolean (PB) constraints we always mean 0-1 integer linear constraints. In what follows, all such constraints are assumed to be written in *normalized form* as non-negative linear combinations of literals $\sum_i a_i \ell_i \geq A$, where the *coefficients* $a_i \in \mathbb{N}_0$ are non-negative integers, the *degree (of falsity)* $A \in \mathbb{N}_+$ is a positive integer, and *literals* ℓ_i represent variables x_i or negated variables $\bar{x}_i$ (which *cancel* to produce $x_i + \bar{x}_i = 1$, and where at most one of x_i and $\bar{x}_i$ appears in any constraint).

A disjunctive clause $x \vee \bar{y} \vee z$ is just a special case $x + \bar{y} + z \geq 1$ of a PB constraint, and we will refer to collections of such constraints as *CNF formulas*. *Cardinality constraints* are another special case where all coefficients are 1 but the degree can be larger. By a *pseudo-Boolean (PB) formula* we mean any collection of (general) PB constraints.

One approach to solving PB formulas is to convert them to CNF, either lazily by learning clauses from PB constraints during conflict analysis, as in one of the version in the *Sat4j* library [Le Berre and Parrain, 2010], or eagerly by re-encoding the whole formula to CNF and running a CDCL solver as in, e.g., *MiniSat+* [Eén and Sörensson, 2006], *Open-WBO* [Martins *et al.*, 2014], or *NaPS* [Sakai and Nabeshima, 2015]. Here we are more interested in solvers doing native pseudo-Boolean reasoning, such as *PRS* [Dixon and Ginsberg, 2002], *Galena* [Chai and Kuehlmann, 2005], *Pueblo* [Sheini and Sakallah, 2006], *Sat4j* [Le Berre and Parrain, 2010], and *RoundingSat* [Elffers and Nordström, 2018] (related, but

[1]A similar idea in the context of constraint satisfaction problems was independently developed in [Bayardo Jr. and Schrag, 1997].

slightly different, ideas were also explored in *bsolo* [Manquinho and Marques-Silva, 2006]). Needless to say, this discussion is far from a complete overview of PB solving or the even richer area of PB optimization—see, e.g., the excellent survey in [Biere *et al.*, 2009, Chapter 22] for more information.

The *cutting planes* proof system [Cook *et al.*, 1987] can be defined as consisting of rules for *literal axioms*

$$\frac{}{\ell_i \geq 0}\ , \tag{1}$$

linear combination

$$\frac{\sum_i a_i\ell_i \geq A \qquad \sum_i b_i\ell_i \geq B}{\sum_i (c_A a_i + c_B b_i)\ell_i \geq c_A A + c_B B} \quad [c_A, c_B \in \mathbb{N}_0]\ , \tag{2}$$

and *division*

$$\frac{\sum_i a_i\ell_i \geq A}{\sum_i \lceil a_i/c \rceil \ell_i \geq \lceil A/c \rceil} \quad [c \in \mathbb{N}_+]\ . \tag{3}$$

A toy example just to illustrate the rules is the derivation

$$\dfrac{\dfrac{6x + 2y + 3z \geq 5 \qquad x + 2y + w \geq 1}{8x + 6y + 3z + 2w \geq 7}\ \text{Linear comb.}}{3x + 2y + z + w \geq 3}\ \text{Division} \tag{4}$$

The setting in this paper is that the input is a PB formula without 0-1 solutions, and the goal is to prove unsatisfiability by deriving $0 \geq 1$. For readers more interested in optimization, this is also the situation when the solver should prove that the objective function cannot be better than in the current solution.

When we want to understand the power of a method of reasoning, we ignore algorithmic aspects and study what can be achieved assuming optimal use of the derivation rules. (This can also be a fruitful perspective because the sophisticated heuristics in modern solvers are typically beyond rigorous analysis.) In this context, it is known that cutting planes is exponentially stronger than the resolution proof system underlying CDCL [Haken, 1985; Cook *et al.*, 1987].

It can be noted that literal axioms and linear combinations are sound also over the reals, so division is where the power of cutting planes lies. In example (4) no information is lost when dividing the constraint, but this does not hold in general—for instance, a further division by 3 would yield the clause $x + y + z + w \geq 1$, which is a strictly weaker constraint.

In conflict-driven solving, linear combinations (2) are always made to cancel some variable on which the two constraints disagree, giving rise to the more restricted *generalized resolution* rule (going back to [Hooker, 1988; 1992])

$$\frac{a_j x_j + \sum_{i \neq j} a_i\ell_i \geq A \qquad b_j \bar{x}_j + \sum_{i \neq j} b_i\ell_i \geq B}{\sum_{i \neq j} \left(\frac{c}{a_j}a_i + \frac{c}{b_j}b_i\right)\ell_i \geq \frac{c}{a_j}A + \frac{c}{b_j}B - c}\ , \tag{5}$$

where $c = \operatorname{lcm}(a_j, b_j)$. What is more, PB solvers based on [Chai and Kuehlmann, 2005] do not use the division rule (3) but instead the *saturation rule*

$$\frac{\sum_i a_i\ell_i \geq A}{\sum_i \min\{a_i, A\} \cdot \ell_i \geq A} \tag{6}$$

saying that no variable coefficient need be larger than the maximum contribution required from that variable. Note that saturation, too, is a "Boolean" rule in that it is not sound over the reals. The derivation

$$\dfrac{\dfrac{2x + y + z \geq 2 \qquad 3\bar{x} + 2y + u + w \geq 3}{7y + 3z + 2u + 2w \geq 6}\ \text{Res. on } x}{6y + 3z + 2u + 2w \geq 6}\ \text{Saturation} \tag{7}$$

shows how resolution and saturation can be combined.

As discussed in [Vinyals *et al.*, 2018], this leads to the following combinations of cutting planes rules to consider from an applied PB solving perspective (where 1(b)+2(b) corresponds to [Cook *et al.*, 1987]):

1. Boolean rule: (a) saturation or (b) division.
2. Linear combinations: (a) resolution or (b) no restrictions.

The use of generalized resolution seems inherent in a conflict-driven context, but which Boolean rule to prefer is less clear. Saturation was used in the seminal paper [Chai and Kuehlmann, 2005] and has also been the rule of choice in what is arguably the most popular PB solver *Sat4j* [Le Berre and Parrain, 2010]. Division appeared only recently in *RoundingSat* [Elffers and Nordström, 2018] (although it was suggested in a more general integer linear programming setting in [Jovanovic and de Moura, 2013]).

But before choosing between these two Boolean rules, it seems natural to ask how they compare in strength! Very little is known about this. [Vinyals *et al.*, 2018] initiated a study of different subsystems of cutting planes, but in the context of PB solving, when linear combinations are restricted to be instances of generalized resolution, they failed to differentiate between division and saturation. This limited understanding stands in striking contrast to the extensive research on different versions of the resolution proof system in the context of CDCL (in, e.g., [Beame *et al.*, 2004; Buss *et al.*, 2008; Atserias *et al.*, 2011; Pipatsrisawat and Darwiche, 2011]).

In this work, we obtain the following results:

1. For cutting planes with saturation, it holds that linear combinations can be restricted to generalized resolution without (any significant) loss of proof power.
2. Cutting planes with division and generalized resolution can be exponentially stronger than cutting planes with saturation and unrestricted linear combinations.
3. To simulate a single combination of generalized resolution plus saturation (as in example (7)) can require a number of division steps that is exponential in the bitsize of the coefficients in the constraints, even if unrestricted linear combinations are allowed.

The first contribution is a strengthening of [Vinyals *et al.*, 2018], which obtained an analogous result when the input is in CNF and all coefficients in the inequalities are restricted to be of at most polynomial magnitude, but as far as we are aware the second and third results are the first of their kind.

As a complement to these theoretical contributions, we also report on a limited empirical evaluation of whether these separations can be observed in practice as well.

The rest of this paper is organized as follows. We present the proofs of the three results listed above in Sections 2, 3, and 4, respectively. After discussing the results from our experiments in Section 5, we make some concluding remarks in Section 6.

2 On the Strength of Generalized Resolution

Let us start by investigating how much of a restriction the generalized resolution rule is. It is known that this can be a severe limitation—on CNF inputs it causes cutting planes to collapse to the much weaker resolution proof system regardless of whether division or saturation is used [Vinyals *et al.*, 2018]. Hence, for cutting planes with division this restriction incurs an exponential loss in strength. However, we show that combined with saturation the generalized resolution rule in fact affords the same power as unrestricted linear combinations.

By way of a quick review of preliminaries, a *cutting planes derivation* π from a PB formula F is a sequence of PB constraints $\pi = (C_1, C_2, \dots, C_L)$ such that each C_i is either from F or is derived from previous constraints using some subset of the rules (1)–(3) and (5)–(6) (depending on the flavour of cutting planes under study). The *length* of a derivation is the number of constraints in it. We say that a derivation π is a *proof (of unsatisfiability) for* F, or *refutation of* F, if $C_L \doteq 0 \geq A$ for $A \in \mathbb{N}_+$ (where $\doteq$ denotes syntactic equality). For the rest of the paper we will use the following terminology:

- A *resolution derivation* is a derivation where (a) the input is in CNF and (b) the only derivation rule is generalized resolution (5) followed by saturation (6) in one step.
- *Cutting planes with division* is the proof system using rules (1)–(3), and a *division derivation (refutation)* is a derivation (refutation) in this proof system.
- *Cutting planes with saturation* is the system with rules (1), (2), and (6) yielding *saturation derivations*.

If the linear combination rule (2) is restricted to be an instance of generalized resolution (5), we say that we have a *division/saturation derivation with (generalized) resolution*. With these conventions we can now state our first theorem.

Theorem 2.1. *If a PB formula F over n variables has a saturation refutation π in length L, then F also has a a saturation refutation π' with generalized resolution in length $\mathrm{O}(n^2 \cdot L)$.*

Furthermore, if F is a CNF formula, then the refutation π' obtained in Theorem 2.1 can be converted to a resolution refutation of F of the same length as π'. To see this, it is sufficient to verify that the degree of falsity in π' can never go above 1, meaning that the constraints are always semantically equivalent to clauses. Note that coefficients larger than 1 are not an issue when the degree is 1—they can simply be viewed as clauses containing the same literal multiple times. Formalizing this argument properly yields the following corollary.

Corollary 2.2. *If a CNF formula F over n variables has a saturation refutation π in length L, then there is a resolution refutation of F in length $\mathrm{O}(n^2 \cdot L)$.*

A weaker form of Corollary 2.2 was shown in [Vinyals *et al.*, 2018], namely with the added (and significant) restriction that all coefficients in the original refutation π have to be small.

In what remains of this section we will prove Theorem 2.1. Starting with the refutation $\pi = (C_1, C_2, \dots, C_L)$, we will construct π' by representing each C_i by a set of m_i constraints $D_{ij} \doteq \sum_{k=1}^{n} a_{ijk}\ell_k \geq A_{ij}$ for $j \in [m_i]$ and associated factors $\delta_{ij} \in \mathbb{N}_+$, writing $\mathcal{D}_i = \{(\delta_{ij}, D_{ij}) \,|\, j \in [m_i]\}$ to denote the constraints and factors for C_i. We will require the following invariants to hold for all $i \in [L]$:

1. $\mathcal{D}_i$ represents C_i in the sense that $\sum_{j=1}^{m_i} \delta_{ij} \cdot D_{ij} = \gamma \cdot C_i$ for some $\gamma \in \mathbb{N}_+$ (where the summation notation denotes taking linear combinations as in (2) but of arbitrary arity).
2. $\mathcal{D}_i$ has size $|\mathcal{D}_i| = m_i \leq n + 1$.
3. Every variable in $\mathcal{D}_i$ occurs with only one *polarity* (i.e., cannot appear both negated and unnegated in $\mathcal{D}_i$).
4. If C_i is derived from $C_{i'}$ (and $C_{i''}$) then all constraints in $\mathcal{D}_i$ can be derived from $\mathcal{D}_{i'}$ (and $\mathcal{D}_{i''}$) using at most $\mathrm{O}(n^2)$ generalized resolution and saturation steps.

Let us argue that Theorem 2.1 follows immediately from a construction maintaining these invariants. The refutation π' will consist of the constraints in the sets $\mathcal{D}_i$ concatenated with the intermediate derivation steps in invariant 4, using only generalized resolution and saturation. Since the final line in the refutation π is $C_L \doteq 0 \geq A$ for some $A \in \mathbb{N}_+$, it follows that all constraints in $\mathcal{D}_L$ are of the form $0 \geq A', A' \in \mathbb{N}_+$ (since adding all constraints in $\mathcal{D}_i$ must yield a multiple of $0 \geq A$ by invariant 1 and no variables can cancel by invariant 3). Finally, the length of π' is $\mathrm{O}(n^2 \cdot L)$ because each constraint C_i is replaced by $n + 1$ constraints by invariant 2 in addition to the $\mathrm{O}(n^2)$ constraints that are used to derive C_i by invariant 4.

We can take care of invariant 2 directly, arguing similarly to the proof of Caratheodory's theorem. We omit the proof due to space constraints, but the idea is that if a positive integer linear combination of a set of constraints $\mathcal{D}$ yields some constraint C, then we only need a linearly independent subset of at most $n + 1$ constraints to get a multiple of C. We now present an inductive construction that maintains the other invariants.

Base Case: $C_i \in F$ or $C_i \doteq \ell \geq 0$. Set $\mathcal{D}_i = \{(1, C_i)\}$. The invariants hold trivially.

Saturation: If C_i is obtained by saturation of $C_{i'}$, which we denote $C_i = \mathsf{sat}(C_{i'})$, then we let $\mathcal{D}_i$ consist of the set $\{(\delta, \mathsf{sat}(D)) \,|\, (\delta, D) \in \mathcal{D}_{i'}\}$ plus possibly $\{(\delta_k, \ell_k \geq 0)\}$ for some literals ℓ_k in $\mathcal{D}_{i'}$ as discussed below. Invariant 3 holds by construction, as it already holds for $\mathcal{D}_{i'}$. Let us argue that Invariants 1 and 4 can be made to hold as well.

First note that if we would sum over $\mathcal{D}_{i'}$ and then saturate, we would obtain the desired constraint $\mathsf{sat}(\sum_{j=1}^{m_{i'}} \delta_{i'j} D_{i'j}) = \mathsf{sat}(\gamma C_{i'})$ using invariant 1 and the fact that $\mathsf{sat}(\gamma C_{i'}) = \gamma \cdot \mathsf{sat}(C_{i'}) = \gamma C_i$, but now we are saturating before taking the summation. However, the degree of falsity in the final constraint does not depend on the order of saturation and summation, as there are no cancellations when adding the constraints in $\mathcal{D}_{i'}$ due to invariant 3, and saturation does not affect the degree. Therefore, the only difference when saturating first are the coefficients, and the only thing that can happen to them is that they get smaller, making the final constraint stronger.

For a fixed literal ℓ_k, the coefficient when saturation happens first is $\sum_{j=1}^{m_i} \delta_{ij} a_{ijk} = \sum_{j=1}^{m_{i'}} \delta_{i'j} \min(a_{i'jk}, A_{i'j}) \leq \min(\sum_{j \in [m_{i'}]} \delta_{i'j} a_{i'jk}, \sum_{j \in [m_{i'}]} \delta_{i'j} A_{i'j})$, where the last expression is the coefficient if summation is done before saturation. Therefore, all that is needed to get $\sum_{j=1}^{m_i} \delta_{ij} D_{ij} = \gamma C_i$ is to add $\ell_k \geq 0$ to $\mathcal{D}_i$ for literals ℓ_k with too small coefficients.

Linear Combination: If C_i is derived by linear combination, i.e., $C_i = cC_{i'} + c'C_{i''}$ then we join the sets $\mathcal{D}_{i'}, \mathcal{D}_{i''}$ and

multiply the factors for each constraint by c or c' to obtain $\mathcal{D}_i' = \{(c\delta, D) \mid (\delta, D) \in \mathcal{D}_{i'}\} \cup \{(c'\delta, D) \mid (\delta, D) \in \mathcal{D}_{i''}\}$. Summing all constraints in $\mathcal{D}_i'$ will yield a multiple of C_i by invariant 1 and because addition of constraints is associative, i.e., the order of addition does not matter.

However, $\mathcal{D}_i'$ might be violating invariant 3. To fix this, for every variable x and every pair of constraints containing x with opposite polarities we can replace one of the constraints by their resolvent over x as in (5). After each such step, it is still true that the constraints in $\mathcal{D}_i'$ can be summed up to yield a multiple of C_i, possibly after adapting the δ-factors. To formalize this argument we need the next lemma.

Lemma 2.3. *Let C_1, C_2 be any two constraints in which a variable x occurs with opposite polarities, and let $\delta_1, \delta_2 \in \mathbb{N}_+$. Then generalized resolution can be used to obtain constraints D_1, D_2 such that x does not occur in D_1 and there are $\gamma, \widehat{\delta}_1, \widehat{\delta}_2 \in \mathbb{N}_0$ for which $\gamma(\delta_1 C_1 + \delta_2 C_2) = \widehat{\delta}_1 D_1 + \widehat{\delta}_2 D_2$.*

Proof. Let a_1, a_2 be the coefficients of x and $\bar{x}$ respectively in C_1, C_2 and let $c = \operatorname{lcm}(a_1, a_2)$, $c_1 = c/a_1$, and $c_2 = c/a_2$. Apply generalized resolution to derive $D_1 = c_1 \cdot C_1 + c_2 \cdot C_2$ (which by construction does not contain x). Assuming without loss of generality (because of symmetry) that $\delta_1 c_2 \geq \delta_2 c_1$, set $D_2 = C_1$, $\widehat{\delta}_1 = \delta_2$, $\widehat{\delta}_2 = \delta_1 c_2 - \delta_2 c_1$, and $\gamma = c_2$. Then it holds that $\gamma \cdot (\delta_1 C_1 + \delta_2 C_2) = (\delta_1 c_2 - \delta_2 c_1) \cdot C_1 + \delta_2 c_1 \cdot C_1 + \delta_2 c_2 \cdot C_2 = \widehat{\delta} D_1 + \widehat{\delta}_2 D_2$, establishing the lemma. □

To restore invariant 3, we apply Lemma 2.3 repeatedly to variables x occurring with opposite polarities in $\mathcal{D}_i'$ as follows. Each time Lemma 2.3 is invoked one constraint (out of at most $n+1$) is replaced by another one that does not contain x. This continues until x occurs with only one polarity or has vanished completely, and this is maintained when the process is repeated for the next variable. Therefore, we will obtain a set $\mathcal{D}_i$ that no longer violates invariant 3 after $\mathrm{O}(n^2)$ applications of the generalized resolution rule.

3 On the Strength of Division

We now turn to studying how the division and saturation rules compare in strength assuming that linear combinations are restricted to generalized resolution, as is the case in conflict-driven PB solving. Without this restriction, [Vinyals *et al.*, 2018] exhibited a family of CNF formulas witnessing that division can be exponentially stronger than saturation in cutting planes. CNF formulas are of no use here, since for such inputs cutting planes with generalized resolution is the same as the resolution proof system regardless of which Boolean rule is used, but nevertheless it is helpful to study these separating CNF formulas. They contain many subsets of clauses

$$\ell_1 + \ell_2 + \ell_3 \geq 1 \tag{8a}$$
$$\ell_1 + \ell_2 + \ell_4 \geq 1 \tag{8b}$$
$$\ell_1 + \ell_3 + \ell_4 \geq 1 \tag{8c}$$
$$\ell_2 + \ell_3 + \ell_4 \geq 1 \tag{8d}$$

which can be summed up to get

$$3\ell_1 + 3\ell_2 + 3\ell_3 + 3\ell_4 \geq 4 \tag{9}$$

after which division by 3 recovers the cardinality constraint

$$\ell_1 + \ell_2 + \ell_3 + \ell_4 \geq 2 \ . \tag{10}$$

Although the constraints (9) and (10) are semantically equivalent over the integers the former constraint is weaker over the reals, and it turns out to be crucial to have constraints of the latter form in order to prove contradiction efficiently.

The reason this yields nothing in a setting with generalized resolution is that there are no literals with opposite polarity in (8a)–(8d), and so there is no legal way to sum up these constraints to give division the chance to go from (9) to (10). However, a moment of thought reveals that we can "cheat" by changing our formula to a "morally equivalent" but syntactically different one. The trick is to re-encode (8a)–(8d) by introducing helper variables x, y and z, writing

$$x + y + z + \ell_1 + \ell_2 + \ell_3 \geq 1 \tag{11a}$$
$$\bar{x} + \ell_1 + \ell_2 + \ell_4 \geq 2 \tag{11b}$$
$$\bar{y} + \ell_1 + \ell_3 + \ell_4 \geq 2 \tag{11c}$$
$$\bar{z} + \ell_2 + \ell_3 + \ell_4 \geq 2 \tag{11d}$$

(where x, y, z are unique to this subset of constraints). Since the helper variables cancel, it is now legal to apply generalized resolution to all constraints. This results in (9), after which division yields the inequality (10) as desired. Applying this re-encoding trick to the separating CNF formulas used in [Vinyals *et al.*, 2018] leads to the following theorem.

Theorem 3.1. *There is a family of PB formulas $\{F_n\}_{n \in \mathbb{N}_+}$ with $\mathrm{O}(n)$ variables and constraints that can be refuted in length $\mathrm{O}(n)$ in cutting planes with division and generalized resolution, but for which any saturation refutations, even with unrestricted linear combinations, have length $\exp(\Omega(n))$.*

Proof. Let $\{F_n\}$ be *subset cardinality formulas* as in [Mikša and Nordström, 2014] that require exponential length for the resolution proof system but that, once cardinality constraints are recovered, have short refutations in cutting planes with generalized resolution as shown in [Vinyals *et al.*, 2018].

Let F_n' be the formula obtained from F_n by introducing helper variables as in (11a)–(11d). We argued above that generalized resolution followed by a division step recovers (10), after which we can use the efficient refutation with generalized resolution in [Vinyals *et al.*, 2018]. It remains to argue why F_n' is exponentially hard for cutting planes with saturation.

Note that if we assign the helper variables to false in (11a)–(11d), then we get back the clauses (8a)–(8d) in the original formula F_n. Letting ρ be the partial assignment, or *restriction*, that sets all helper variables in the whole formula to false, we write $C\restriction_\rho$ for the result of applying ρ to a constraint C, and extend this notation to sets of constraints by taking unions. It is not hard to show that if π is a saturation (or division) refutation of F with unrestricted linear combinations, then applying ρ to the lines of π results in a saturation (or division, respectively) refutation $\pi\restriction_\rho$ of $F\restriction_\rho$, except that we might need to insert some linear combinations with literal axioms to make sure that the derivation stays syntactically valid (but this can only increase the length by a factor of n.)

Let π' be a saturation refutation of F_n'. Applying ρ yields a saturation refutation $\pi'\restriction_\rho$ of $F_n'\restriction_\rho = F_n$ that is at most

a factor $\mathrm{O}(n)$ longer. Appealing to Corollary 2.2, we obtain a resolution refutation π^* at most a factor $\mathrm{O}(n^2)$ longer than $\pi'\restriction_\rho$. But by [Mikša and Nordström, 2014] we know that any resolution refutation π^* of F_n must have exponential length, and hence an exponential lower bound holds also for the saturation refutation π' of F'_n. The theorem follows. □

4 On the Strength of Saturation

So far in the paper we have given evidence that the division rule can be exponentially stronger than the saturation rule in certain contexts. In this section we show that there are settings in which saturation can be significantly stronger than division. Unfortunately, we are not able to get a analogous result to Theorem 3.1, but what we can prove is that in order to go from

$$C_1(R) \doteq Rx + Ry + \textstyle\sum_{j=1}^{R} z_j \geq R \tag{12a}$$

$$C_2(R) \doteq Rx + R\overline{y} + \textstyle\sum_{j=R+1}^{2R} z_j \geq R \tag{12b}$$

to

$$C_L(R) \doteq Rx + \textstyle\sum_{j=1}^{2R} z_j \geq R \ , \tag{13}$$

which can be done with one resolution step followed by one saturation step, at least $\Omega\big(\sqrt{R}\big)$ applications of the division rule are required (in addition to other steps). Note that this is exponential in the bitsize of R.

A formal proof follows below, but let us first sketch the idea. It can be shown by a simple inductive argument that for any constraint containing negated literals $\overline{x}$ or $\overline{z}_j$ we can instead derive the same constraint without these literals. Therefore, it is only necessary to consider constraints C_i of the form

$$a_i x + b_i y + c_i \overline{y} + \textstyle\sum_{j=1}^{2R} d_{ij} z_j \geq A_i \tag{14}$$

(where $\min\{b_i, c_i\} = 0$). For such a constraint C_i we write $B_i = 2a_i + b_i + c_i$ and define the *potential* $\mathcal{P}(C_i)$ to be

$$\mathcal{P}(C_i) = \ln\big(B_i/A_i\big) \ . \tag{15}$$

Note that $\mathcal{P}(C_1(R)) = \mathcal{P}(C_2(R)) = \ln(3) > \ln(2) = \mathcal{P}(C_L(R))$. What we will prove in Lemma 4.2 is that only division can decrease the potential of derived constraints, and only does so by an amount of at most $1/\sqrt{R}$. This shows that $\Omega(\sqrt{R})$ divisions are required to derive $C_L(R)$ from $C_1(R)$ and $C_2(R)$. Our formal result is as follows.

Theorem 4.1. *Let $R = K^2$ for $K \in \mathbb{N}_+$ and let π be a division derivation of $C_L(R)$ from $C_1(R)$ and $C_2(R)$ (with unrestricted linear combinations). Then π contains $\Omega(\sqrt{R})$ applications of the division rule.*

We remark that the lower bound is tight except possibly for the square root. As discussed in [Vinyals *et al.*, 2018], for constraints with coefficients of size at most R it is always possible to simulate saturation with $\mathrm{O}(R)$ division and unrestricted linear combination steps.

To establish Theorem 4.1, we start with a preprocessing step to ensure that the degree A_i of any constraint $C_i \in \pi$ obtained by division is sufficiently large, namely $A_i \geq \sqrt{R} + 1$.

Suppose $A_{i_1} < \sqrt{R} + 1$ for some constraint C_{i_1} resulting from division. Since $\sqrt{R}$ is an integer by assumption, we have $A_{i_1} \leq \sqrt{R}$. We claim that C_{i_1} can be satisfied by setting at most $\sqrt{R}$ variables z_j to true. To see why, note that $C_1(R)$ and $C_2(R)$ are satified by setting all z_j to true, and hence any constraint derived from them must also be satisfied by this assignment since the proof system is sound. Furthermore, for C_{i_1} it must be sufficient to assign a subset of at most $\sqrt{R}$ variables z_j, since every z_j contributes at least 1 to the left-hand side and the degree on the right is $A_{i_1} \leq \sqrt{R}$. Let ρ_1 be such a partial assignment to at most $\sqrt{R}$ variables z_j fixing C_{i_1} to true and consider the restricted derivation $\pi\restriction_{\rho_1}$ (where $C_{i_1}\restriction_\rho$ has been removed since it is now a trivial constraint).

Suppose the derivation $\pi\restriction_{\rho_1}$, contains some constraint C_{i_2} with degree $A_{i_2} < \sqrt{R} + 1$ (note that degrees might have decreased after the restriction ρ_1). Argue as above to find a restriction ρ_2 to at most $\sqrt{R}$ variables z_j satisfying C_{i_2}, and continue with the derivation $\pi\restriction_{\rho_1 \cup \rho_2}$. We repeat this procedure for T steps if possible as long as $T \leq \sqrt{R}/6$. If at the end of this process there is still some constraint C_i with degree $A_i < \sqrt{R} + 1$, then we have counted $\sqrt{R}/6$ division steps, which is enough to obtain the lower bound in Theorem 4.1. Otherwise, it now holds for $\rho = \rho_1 \cup \cdots \cup \rho_T$ that all constraints in $\pi\restriction_\rho$ have degree $A_i \geq \sqrt{R} + 1$ and that ρ assigns at most $(\sqrt{R}/6) \cdot \sqrt{R} = R/6$ variables z_j.

For the rest of the proof we will focus on the restricted derivation $\pi\restriction_\rho$. It is immediate from (15) that the potential of the constraints can only increase compared to π, since restricting variables can only cause the degree of falsity to go down. Hence, for the initial constraints we have $\min\{\mathcal{P}(C_1(R)\restriction_\rho), \mathcal{P}(C_2(R)\restriction_\rho)\} \geq \ln(3)$. But the potential does not increase too much—since ρ sets at most $R/6$ variables z_j, for the final constraint we have $\mathcal{P}(C_L(R)\restriction_\rho) \leq \ln(2R/(R - R/6)) = \ln(12/5)$. Therefore, the difference is still at least $\ln(3) - \ln(12/5) = \ln(15/12) \geq 1/6$. We will now show that the potential can only decrease after a division step, and only by $1/\sqrt{R}$, which establishes Theorem 4.1.

For convenience, we split the linear combination rule (2) into two rules for multiplying a constraint and adding two constraints. Also, when dividing a constraint C by $k \in \mathbb{N}_+$ as in (3), which we denote $\mathsf{div}(C, k)$, we assume k divides all coefficients in C. This is without loss of generality, since literal axioms (1) can be added as needed to make this true.

Lemma 4.2. *For any constraints C_i and $C_{i'}$ of the form* (14) *derived from* (12a) *and* (12b) *(possibly with some variables z_j restricted to true), for any literal axiom E as in* (1)*, and for any $k \in \mathbb{N}_+$, it holds that:*

1. $\mathcal{P}(k \cdot C_i) = \mathcal{P}(C_i)$.
2. $\mathcal{P}(C_i + k \cdot E) \geq \mathcal{P}(C_i)$.
3. $\mathcal{P}(C_i + C_{i'}) \geq \min\{\mathcal{P}(C_i), \mathcal{P}(C_{i'})\}$.
4. $\mathcal{P}(\mathsf{div}(C_i, k)) \geq \mathcal{P}(C_i) - 1/\sqrt{R}$, *assuming that* $\mathsf{div}(C_i, k)$ *has degree at least* $\sqrt{R} + 1$.

Proof. Part 1 is obvious from the definition in (15).

Part 2 is trivially true if no cancellation occurs as only the numerator in the potential can increase. Suppose that the degree decreases by $k' \leq k$ through cancellation on x.

Analogously to the argument in the preprocessing step, setting x to true satisfies $C_1(R)$ and $C_2(R)$ and hence also C_i. Thus, $a_i \geq A_i$ and $B_i \geq 2 \cdot A_i$, from which it follows that $\exp(\mathcal{P}(C_i + k \cdot E)) = (B_i - 2k')/(A_i - k') \geq B_i/A_i = \exp(\mathcal{P}(C_i))$. It is straightforward to verify that if the cancellation is due to some other variable than x, then the numerator can only be larger, and hence so will the potential.

For part 3, assuming that $\mathcal{P}(C_i) \leq \mathcal{P}(C_{i'})$ (without loss of generality due to symmetry), we have $(B_i + B_{i'})/(A_i + A_{i'}) \geq B_i/A_i$. As in part 2, we have $B_i \geq 2 \cdot A_i$ and $B_{i'} \geq 2 \cdot A_{i'}$. Let k be the decrease in degree due to cancellation on y (other variables do not occur negated in (14) and cannot cancel). Then we get $\exp(\mathcal{P}(C_i + C_{i'})) = (B_i + B_{i'} - 2k)/(A_i + A_{i'} - k) \geq (B_i + B_{i'})/(A_i + A_{i'}) \geq B_i/A_i = \exp(\mathcal{P}(C_i))$.

For part 4, since all coefficients are divisible by k this also holds for B_i. Therefore, $\mathcal{P}(C_i) - \mathcal{P}(\mathsf{div}(C_i, k)) = \ln(B_i/A_i) - \ln\left((B_i/k)/\lceil A_i/k \rceil\right) = \ln\left(\lceil A_i/k \rceil/(A_i/k)\right) \leq \ln\left((A_i/k + 1)/(A_i/k)\right) \leq k/A_i \leq 1/\sqrt{R}$, where the second to last inequality is just $\ln(1+x) \leq x$ and the last inequality holds since $A_i/k \geq \lceil A_i/k \rceil - 1 \geq \sqrt{R}$ by the assumption about the degree $\lceil A_i/k \rceil$ of $\mathsf{div}(C_i, k)$. □

5 Empirical Evaluation

We have shown that division and saturation are incomparable in strength as rules of reasoning. It is important to understand, however, that these results speak only about the *existence* of proofs and not about whether pseudo-Boolean solvers will actually be able to *find* such proofs. Although our main focus in this paper is on the former question, in this section we report on some limited experiments to shed some light on the latter.

We have run instrumented versions of *Sat4j*, which uses saturation, and *RoundingSat*, which defaults to division but has an option to use saturation instead. All experiments were performed on 4 AMD Opteron 6238 (Interlagos) 12-core 2.6 GHz processors with 128 GB RAM with a 5000-second time-out.

The heuristics for PB solvers are not at all as well-tuned as those for CDCL solvers, and small changes in internal settings can have huge, and currently not so well understood, effects on performance. In order to measure the overall impact of division versus saturation—rather than of some other, unrelated settings—we have therefore run the solvers with several different parameter settings and measured for each PB instance the best result with division and saturation, thus obtaining *virtual best solvers (VBS)* for division and saturation, respectively. More details about the experiments and full results can be found at www.csc.kth.se/~jakobn/DivisionVsSaturation.

To obtain benchmarks that are easy for division but hard for saturation, we use subset cardinality formulas as in Section 3, generated from 4-regular random bipartite graphs with an additional random edge added (see [Mikša and Nordström, 2014] for more details). Almost all the constraints in these formulas are of the form $\ell_1 + \ell_2 + \ell_3 + \ell_4 \geq 2$, and we have compared solver performance on this "unobfuscated" version with the "division-friendly" version based on the clausal encoding in (8a)–(8d), enhanced with helper variables as in (11a)–(11d). We have run the solvers on instances of increasing size to see how the performance scales asymptotically.

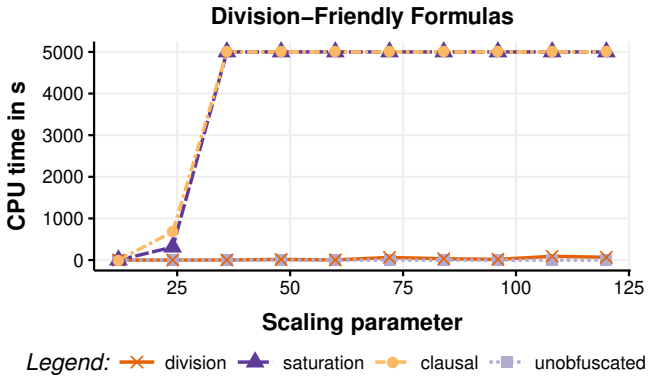


Figure 1: Virtual best division/ saturation solver for division-friendly formula compared to clausal and unobfuscated encoding.

As shown in Figure 1, we have consistently very poor solver performance for the clausal encoding. This is as expected, since this version is exponentially hard for both division- and saturation-based solvers. Furthermore, while the unobfuscated encoding is easy for both division and saturation, for the division-friendly encoding the saturation-VBS again struggles whereas the division-VBS is able to harness the helper variables to achieve a performance close to that of the unobfuscated encoding. This all fits perfectly with theory. Before we get too carried away by this, however, it should be noted that this result is rather fragile. Whether the division-based solver works well or not depends heavily also on how other internal parameters are adjusted, and it turns out to be even more crucial exactly how the helper variables are added.

It is not known whether there exist PB formulas that are easy for saturation but provably hard for division—this is a very interesting question left open by our work. What we can do to obtain interesting benchmarks, though, is to use inspiration from Section 4 to design formulas that are easy for saturation but *appear* to be tricky for division. To this end, we construct a pigeonhole principle-like formula which we think of as being defined in terms of a $(2R+2) \times (2R+1)$ matrix, where we scale R to increase the instance size. The variables are x_{ij}, $i \in [2R+2]$, $j \in [2R+1]$, with coefficients $a_{ii} = R$ and $a_{ij} = 1$ for $i \neq j$. We first consider an "unobfuscated" formula with constraints

$$\sum_{j=1}^{2R+1} a_{ij} x_{ij} \geq R \qquad \text{for } i \in [2R+2] \tag{16a}$$

$$\sum_{i=1}^{2R+2} a_{ij} x_{ij} \leq R \qquad \text{for } j \in [2R+1] \tag{16b}$$

which is easily seen to be unsatisfiable by adding all row constraints (16a) and all column constraints (16b) separately. This proof can be carried out using only generalized resolution, and so these formulas are easy in theory for all PB solvers regardless of which Boolean rule they use.

To get a formula that is easy for saturation but potentially tricky for division, we observe that for the rows $i < 2R+2$ the constraint (16a) is of the form (13) and can be "split" into

$$Rx_{i,i} + Ry_i + \sum_{j=1;j\neq i}^{R'_i} a_{ij} x_{ij} \geq R \tag{17a}$$

$$Rx_{i,i} + R\overline{y}_i + \sum_{j=R'_i+1;j\neq i}^{2R+1} a_{ij} x_{ij} \geq R \tag{17b}$$

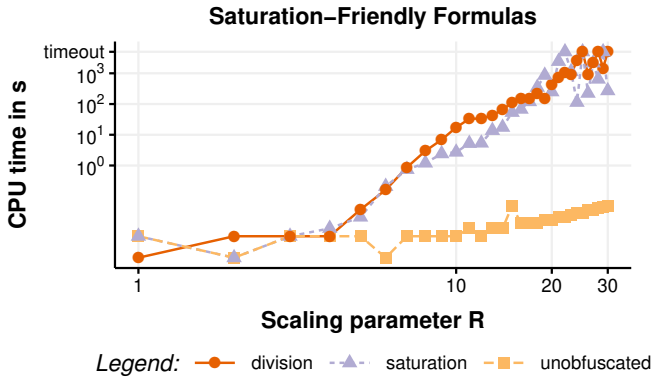


Figure 2: Virtual best division/ saturation solver for saturation-friendly formula compared to unobfuscated encoding as baseline

similar to (12a)–(12b), where $R'_i = R + 1$ if $i \leq R$ and $R'_i = R$ if $i > R$. It is straightforward to verify that this does not change anything from a saturation point of view—the new formula can still be solved in $\mathrm{O}(R)$ steps by first using generalized resolution plus saturation to go from (17a) and (17b) back to (16a), reverting the obfuscation, and then adding all constraints (16a) and (16b) in a cancelling way as discussed above. If we use division instead of saturation, the formula can be solved in $\mathrm{O}\big(R^2\big)$ steps, since each recovery of a constraint (16a) from (17a)–(17b) can be done with at most R divisions and linear combinations [Vinyals *et al.*, 2018], but we know from Section 4 that any proof starting by such a "recovery phase" requires $\Omega\big(R \cdot \sqrt{R}\big) = \Omega\big(R^{3/2}\big)$ steps. Furthermore, it should be noted that the refutation in length $\mathrm{O}\big(R^2\big)$ employs non-cancelling linear combinations, and it is not clear what the best approach is if we insist on using generalized resolution, as PB solvers do. Thus, we could hope that formulas of this type should be significantly harder for solvers using division than for solvers using saturation.

Sadly, however, the experimental results fail to confirm this intuition. The problem is not that these formulas are too easy for division, but rather that they are too hard for saturation (see Figure 2). While the baseline version is easy as expected, the obfuscated formula is hard for both division and saturation with no clear difference between the two.

This finding illustrates what we discussed at the beginning of this section, namely the difference between the non-constructive existence of short proofs and the constructive, algorithmic search for such short proofs. In this case, not only the choice of rules is important, but also the order in which these rules are applied. To illustrate this, let $R = 2$ and consider, e.g., the "split" row constraint (17a)–(17b) for $i = 1$ and the column constraint (16b) for $j = 2$, i.e.,

$$2x_{11} + 2y_1 + x_{12} + x_{13} \geq 2 \tag{18a}$$
$$2x_{11} + 2\bar{y}_1 + x_{14} + x_{15} \geq 2 \tag{18b}$$
$$\bar{x}_{12} + 2\bar{x}_{22} + \bar{x}_{32} + \bar{x}_{42} \geq 3 \tag{18c}$$

(where (18c) is just (16b) written in normalized form). Performing generalized resolution on (18a) and (18b) followed by saturation recovers the unobfuscated row constraint

$$2x_{11} + x_{12} + x_{13} + x_{14} + x_{15} \geq 2 \ , \tag{19}$$

and resolving this constraint with (18c) yields

$$2x_{11} + x_{13} + x_{14} + x_{15} + 2\bar{x}_{22} + \bar{x}_{32} + \bar{x}_{42} \geq 4 \ . \tag{20}$$

If we instead apply the resolution rule on (18c) with (18a) and then resolve the resulting constraint with (18b), we get

$$4x_{11} + x_{13} + x_{14} + x_{15} + 2\bar{x}_{22} + \bar{x}_{32} + \bar{x}_{42} \geq 4 \ . \tag{21}$$

Note that the difference between (20) and (21) is that the coefficient of x_{11} is 4 instead of 2 in the latter, resulting in a strictly weaker constraint.

It is possible to force a saturation-based solver to find short proofs for formulas with obfuscated constraints (17a)–(17b) (in particular, by hard-coding a specific decision order for the variables), but this is nothing that a solver with default heuristics is currently able to do. This suggests that in addition to carefully choosing the set of derivation rules, optimizing the order in which these rules are applied during search is an important part of improving pseudo-Boolean solvers further.

6 Concluding Remarks

In this work we study the relative strength of division and saturation in pseudo-Boolean reasoning. We show that there are formulas for which PB solvers using division can be exponentially faster than solvers using saturation. In the other direction, we prove that the number of division steps needed to simulate a single saturation step can be exponential, but leave open the question of whether saturation-based solvers can ever be strictly stronger than division-based solvers.

By necessity, the formulas we use to obtain these results are crafted so as to be amenable to rigorous mathematical analysis. It would be nice to find more natural benchmarks, and also to study whether the difference in reasoning power between division and saturation ever comes into play in an applied context. Our limited experiments on crafted benchmarks indicate that other aspects of the search heuristics can easily become more important, but this also points to room for improvement of these heuristics (perhaps by, e.g., adaptively choosing between, or combining, division and saturation).

Acknowledgements

We are grateful to Jan Elffers and Marc Vinyals for extensive and enlightening discussions about cutting planes and pseudo-Boolean solving. We would also like to thank the anonymous reviewers for *IJCAI* and *Pragmatics of SAT* for several insightful comments that helped improve the exposition considerably.

Part of this work was carried out while the second and third authors visited the Simons Institute for the Theory of Computing in association with the DIMACS/Simons Collaboration on Lower Bounds in Computational Complexity, which is conducted with support from the National Science Foundation. Our computational experiments used resources provided by the Swedish National Infrastructure for Computing (SNIC).

The first and second authors were funded by the Swedish Research Council (VR) grant 2016-00782. The second author was also supported by the Knut and Alice Wallenberg grant KAW 2016.0066 and the VR grant 621-2012-5645. The third author was partially supported by the ISF grant 1162/15.

References

[Atserias *et al.*, 2011] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, 2011.

[Bayardo Jr. and Schrag, 1997] 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, 1997.

[Beame *et al.*, 2004] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[Biere *et al.*, 2009] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Buss *et al.*, 2008] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL-algorithms with clause learning. *Logical Methods in Computer Science*, 4(4:13), 2008.

[Chai and Kuehlmann, 2005] 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, 2005.

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

[Cook, 1971] 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, 1971.

[Dixon and Ginsberg, 2002] 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, 2002.

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

[Elffers and Nordström, 2018] 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, 2018.

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

[Hooker, 1988] John N. Hooker. Generalized resolution and cutting planes. *Annals of Operations Research*, 12(1):217–239, 1988.

[Hooker, 1992] John N. Hooker. Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6(1):271–286, 1992.

[Jovanovic and de Moura, 2013] Dejan Jovanovic and Leonardo de Moura. Cutting to the chase solving linear integer arithmetic. *Journal of Automated Reasoning*, 51(1):79–108, 2013.

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

[Manquinho and Marques-Silva, 2006] Vasco M. Manquinho and João Marques-Silva. On using cutting planes in pseudo-Boolean optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:209–219, 2006.

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

[Martins *et al.*, 2014] 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, 2014.

[Mikša and Nordström, 2014] Mladen Mikša and Jakob Nordström. Long proofs of (seemingly) simple formulas. 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 121–137. Springer, 2014.

[Moskewicz *et al.*, 2001] 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, 2001.

[Pipatsrisawat and Darwiche, 2011] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.

[Sakai and Nabeshima, 2015] 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, 2015.

[Sheini and Sakallah, 2006] 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, 2006.

[Vinyals *et al.*, 2018] Marc Vinyals, Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, and Jakob Nordström. In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving. 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 292–310. Springer, 2018.