27th International Conference on Theory and Applications of Satisfiability Testing

SAT 2024, August 21–24, 2024, Pune, India

Edited by

Supratik Chakraborty
Jie-Hong Roland Jiang

LIPICS

LIPIcs – Vol. 305 – SAT 2024 www.dagstuhl.de/lipics

Editors

Supratik Chakraborty
IIT Bombay, India
supratik@cse.iitb.ac.in

Jie-Hong Roland Jiang
National Taiwan University, Taipei, Taiwan
jhjiang@ntu.edu.tw

ACM Classification 2012
Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Hardware → Hardware validation

ISBN 978-3-95977-334-8

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-334-8.

Publication date
August, 2024

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.SAT.2024.0

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface

This volume contains (extended) abstracts of invited talks and full papers presented at the 27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024). SAT 2024 was held from August 21–24, 2024, in the Sahyadri Park facility of Tata Consultancy Services in Pune, India. This was the first time that the conference was held in India, and it was jointly organized by Indian Institute of Technology Bombay and Tata Consultancy Services.

The International Conference on Theory and Applications of Satisfiability Testing (SAT) is the topmost annual meeting focused on all aspects of the propositional satisfiability problem, broadly construed. In addition to propositional satisfiability in its classical form, the scope of the conference also includes Boolean optimization problems like MaxSAT and Pseudo-Boolean (PB) constraint solving, Quantified Boolean Formulas (QBF), Satisfiability Modulo Theories (SMT), Model Counting, and Constraint Programming (CP) for problems with clear connections to Boolean-level reasoning.

The call for papers for SAT 2024 welcomed original contributions addressing different aspects of SAT, interpreted in a broad sense. This included theoretical advances such as exact algorithms, proof complexity, and other complexity issues, practical search algorithms, knowledge compilation, implementation-level details of SAT solvers and SAT-based systems, problem encodings and reformulations, applications including both novel application domains and improvements to existing approaches, as well as case studies and reports on findings based on rigorous experimentation. Both regular (long) papers and short papers were welcomed for submission, with the latter format also including tool papers.

A total of 70 submissions, with authors distributed across 25 countries, were received by the submission deadline. Of these, 52 were regular papers, 4 were short papers, and 14 were tool papers. Each submission was reviewed by at least 3 (4 in most cases) Program Committee members and external expert reviewers identified by the Program Committee. Authors of submitted papers were provided an opportunity to respond to initial reviews and to specific queries posed by reviewers during an author response period that spanned 5 days. Subsequent to this, there was extensive discussion among Program Committee members for finalizing the list of accepted papers. Eventually, 24 papers were unconditionally accepted, and an additional 4 papers were accepted subject to the condition that the authors addressed specific comments of reviewers, and the revised version went through a fresh round of review. Authors of conditionally accepted papers were given 2 weeks to revise their papers. Each revised paper was again reviewed by two of the reviewers who had originally reviewed the paper. The 4 conditionally accepted papers were eventually accepted after the reviewers satisfied themselves that all reviewers' comments were attended to adequately in the revised versions of the papers. Overall, a total of 28 papers were finally accepted for SAT 2024, with 23 of these being regular papers, 4 being tool papers, and 1 being a short paper.

In addition to the presentation of accepted papers, the technical program of SAT 2024 also included three invited talks by eminent researchers in the area. Martina Seidl (Institute for Formal Models and Verification at Johannes Kepler University, Austria), Cesare Tinelli (The University of Iowa, USA), and Moshe Y. Vardi (Rice University, USA) delivered invited talks on important and contemporary topics of significant interest to the community. Abstracts or extended abstracts of their talks are included as part of this volume.

In recognition of outstanding and impactful contributions in areas of relevance to the conference, four awards were presented at SAT 2024. The Program Committee deliberated and gave suggestions for the choice of the Best Paper and the Best Student Paper, the latter for a paper whose lead author was a student when the work was done. The Award Committee, consisting of Nikolaj Bjørner, Shaowei Cai, Daniel Le Berre, and Jakob Nordström, carefully considered these suggestions to arrive at the final award recommendations. Accordingly, the **Best Paper Award** of SAT 2024 was conferred on "The Strength of the Dominance Rule" by Leszek Aleksander Kołodziejczyk and Neil Thapen *"for the use of theoretical tools from mathematical logic to answer a question of considerable applied interest regarding the strength of different methods for proof logging."* The **Best Student Paper Award** of SAT 2024 was conferred on "Speeding-up Pseudo-Boolean Propagation" by Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez Carbonell, and Rui Zhao (student author) *"for the principled approach used for measuring the impact of implementation choices for Pseudo-Boolean propagation."*

The three runner-ups for the above awards are: "Optimal Layout Synthesis for Deep Quantum Circuits on NISQ Processors with 100+ Qubits" by Irfansha Shaik and Jaco van de Pol, "Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving" by Jiong Yang, Yaroslav A. Kharkov, Yunong Shi, Marijn Heule, and Bruno Dutertre, and "eSLIM: Circuit Minimization with SAT Based Local Improvement" by Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider.

This year, the SAT Association also established a new award called the ***Fahiem Bacchus PhD Award in Satisfiability***. This award distinguishes one outstanding PhD thesis from the past two years in the field of satisfiability, broadly construed. The award is named after Fahiem Bacchus, who made significant contributions to the theory and practice of SAT, to the SAT community, and to the SAT Association. In response to the call for nominations, the committee for selecting the Fahiem Bacchus Award received a number of strong applications.

The inaugural Fahiem Bacchus Award was conferred at SAT 2024 on Dominik Schreiber for his PhD thesis "Scalable SAT Solving and its Applications", defended at the Karlsruhe Institute of Technology (KIT) in 2023.

The two runner-ups for the award are: Stephan Gocht (Lund University, thesis title: "Certifying Correctness for Combinatorial Algorithms by Using Pseudo-Boolean Reasoning") and André Schidler (Technische Universität Wien, thesis title: "Scalability for SAT-based Combinatorial Problem Solving").

The ***SAT Test-of-Time Award*** is an award instituted by the SAT Association that is given annually to a selected paper from around 20 years back with a large impact on SAT and beyond. The SAT 2024 Test-of-Time Award was conferred on the paper "Combining Component Caching and Clause Learning for Effective Model Counting", which appeared in the proceedings of SAT 2004 and was authored by Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi.

The week of SAT 2024 also included a pre-conference instructional school and three workshops that were held at the same venue as the main conference. The 9^{th} edition of the *Indian SAT+SMT School*, organized by Supratik Chakraborty, Priyanka Golia, Ashutosh Gupta, Saurabh Joshi, Kumar Madhukar, and Kuldeep S. Meel, was held during August 18–19, 2024. Subsequently, the following three workshops were concurrently held on August 20, 2024.

- *Pragmatics of SAT*, organized by Katalin Fazekas and Alexey Ignatiev
- *International Workshop on Quantified Boolean Formulas and Beyond*, organized by Hubie Chen, Friedrich Slivovsky, and Martina Seidl
- *Workshop on Counting, Sampling and Synthesis*, organized by Paulius Dilkas and Priyanka Golia

The results of four major competitions, as listed below, were also announced during the technical program of SAT 2024.

- *SAT Competition 2024*, organized by Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda
- *Pseudo-Boolean Competition 2024*, organized by Olivier Roussel
- *Model Counting Competition 2024*, organized by Johannes Fichte, Markus Hecher, and Arijit Shaw
- *MaxSAT Evaluation 2024*, organized by Matti Järvisalo, Jeremias Berg, Ruben Martins, Tobias Paxian, and Andreas Niskanen

We are thankful to all individuals, institutions, and organizations who contributed to making SAT 2024 a success. We thank all authors for submitting their work to SAT 2024, and also thank all instructional school, workshop, and conference participants for actively participating in the technical sessions. Our deepest thanks go to all Program Committee members and external expert reviewers for carefully reading the submissions, providing constructive comments, and for participating in extensive discussions that helped in selecting the strongest papers for the technical program of the conference. A special thanks to the Award Committee for working on a very tight schedule to arrive at the Best Paper and Best Student Paper decisions, and also for recommending runner-ups for these awards. We are thankful to the three invited speakers for accepting our invitation and enlightening the audience with their insights and perspectives. Our thanks to the organizers of all the school, workshops, and competitions that were affiliated to SAT 2024. A special thanks to Daniela Kaufmann and Alexey Ignatiev for serving as Workshops Chair of SAT 2024.

We are thankful to R. Venkatesh, who served as General Chair of SAT 2024, and to the entire local organizing committee that worked tirelessly under the leadership of Ravindra Metta to make the conference a success. We wish to thank Finance Chair Hrishikesh Karmarkar, Sponsorship Chair Kumar Madhukar, Publicity Chair Abhisekh Sankaran, Advisory Chair Kuldeep S. Meel, and Webmasters Weichieh Wang and Anand Yeolekar for their crucial roles in various aspects of organization of the conference. Our special thanks go to Indian Institute of Technology Bombay and Tata Consultancy Services for helping us negotiate mandatory clearance procedures for SAT 2024. We thank the SAT Association for its invaluable guidance and support at all steps of organizing the conference. Thanks are also due to the EasyChair conference management system for facilitating the submission, review, and selection of papers, and to the editorial and publishing staff at LIPIcs for their help and guidance in compiling this volume.

Finally, we wish to thank all sponsors and supporters of SAT 2024, whose generous financial support made it possible to organize the conference smoothly. We gratefully acknowledge the enormous support, both financial and logistical, provided by Tata Consultancy Services. In addition, sponsorship was provided by Amazon Web Services, Cadence Design Systems, Synopsys, Advanced Micro Devices, Google, and Microsoft. We are grateful to Dr. Anand Deshpande for his support, advice, and encouragement. Generous support for international student travel grants was provided by the journal of Artificial Intelligence (AIJ), while similar support for travel and registration of Indian students was provided by the ACM India Council and the Association for Logic in India. The ACM India Council and the Indian Association for Research in Computing Science provided invaluable support at all stages of organizing the conference. Our sincere thanks to all these organizations for their role in making SAT 2024 a success.

August 2024

Supratik Chakraborty
Jie-Hong Roland Jiang

Program Committee Members

Carlos Ansòtegui	University Of Lleida
Jeremias Berg	University of Helsinki
Olaf Beyersdorff	Friedrich Schiller University Jena
Armin Biere	Albert Ludwigs University Freiburg
Nikolaj Bjørner	Microsoft Research
Shaowei Cai	Institute of Software, Chinese Academy of Sciences
Supratik Chakraborty	IIT Bombay (co-chair)
Katalin Fazekas	TU Wien
Vijay Ganesh	Georgia Institute of Technology
Marijn Heule	Carnegie Mellon University
Alexey Ignatiev	Monash University
Markus Iser	Karlsruhe Institute of Technology
Mikoláš Janota	Czech Technical University in Prague
Jie-Hong Roland Jiang	National Taiwan University (co-chair)
Matti Järvisalo	University of Helsinki
Daniela Kaufmann	TU Wien
Benjamin Kiesl-Reiter	Amazon Web Services, Inc.
Oliver Kullman	Swansea University
Daniel Le Berre	CNRS – Université d'Artois
Jordi Levy	IIIA-CSIC
Inês Lynce	INESC-ID/IST, Universidade de Lisboa
Meena Mahajan	The Institute of Mathematical Sciences, Chennai
Ruben Martins	Carnegie Mellon University
Kuldeep S. Meel	University of Toronto
Stefan Mengel	CNRS, CRIL
Alexander Nadel	Technion & Intel
Aina Niemetz	Stanford University
Jakob Nordström	University of Copenhagen and Lund University
Luca Pulina	University of Sassari
Kristin Yvonne Rozier	Iowa State University
Christoph Scholl	University of Freiburg
Roberto Sebastiani	University of Trento
Natasha Sharygina	University of Lugano
Laurent Simon	Bordeaux Institute of Technology
Friedrich Slivovsky	Vienna University of Technology
Mate Soos	Ethereum Foundation
Martin Suda	Czech Technical University in Prague
Stefan Szeider	TU Wien
Marc Vinyals	University of Auckland

External Reviewers

Martin Blicha
Filip Bàrtek
Florent Capelli
Abhimanyu Choudhury
Jan Dreier
Grigory Fedyukovich
Mathias Fleury
Thibault Gauthier
Filipe Gouveia
Tim Hoffmann
Jan Hůla
Christoph Jabs
Prithwish Jana
Piyush Jha
Markus Kirchweger
Faezeh Labbaf
Ian Li
Gabriele Masina
Alan Mishchenko
Paolo Morettin
Sasank Mouli
Andreas Niskanen
Pedro Orvalho
Shuo Pang
Tobias Paxian
Jelle Piepenbrock
Franz Reichl
Alec Rosentrater
Agnes Schleitzer
Luc Spachmann
Bernardo Subercaseaux
Romain Wallon
Amalee Wilson
Emre Yolcu
Konstantin Britikov
Benjamin Böhm
Leroy Chew
Karel Chvalovskỳ
Clemens Eisenhofer
Margarida Ferreira
Nils Froleyks
Yassine Ghannane
Bernhard Gstrein
Antti Hyvärinen
Hannes Ihalainen
Jan Jakubuv
Duri Andrea Janett
Kaspar Kasche
Tomáš Kolárik
Brian Li
John Lu
Carlos Mencía
Neeldhara Misra
Antonio Morgado
Sibylle Möhle
Andy Oertel
Rodrigo Otoni
Davide Pastorello
Tomáš Peitl
Joseph Reeves
Andrew Reynolds
Prakash Saivasan
Dmitry Sokolov
Giuseppe Spallitta
Miguel Terra-Neves
Zili Wang
Suwei Yang

List of Authors

Markus Anders (4)
TU Darmstadt, Germany

Olaf Beyersdorff (5)
Friedrich Schiller University Jena, Germany

Armin Biere (6, 21)
University Freiburg, Germany

Ilario Bonacina (7)
Universitat Politècnica de Catalunya, Barcelona, Spain

Maria Luisa Bonet (7)
Universitat Politècnica de Catalunya, Barcelona, Spain

Sofia Brenner (4)
TU Darmstadt, Germany

Shaowei Cai (8)
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China;
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Yi Chu (8)
Institute of Software, Chinese Academy of Sciences, Beijing, China

Robin Coutelier (9)
TU Wien, Austria

Yogesh Dahiya (10)
The Institute of Mathematical Sciences (A CI of Homi Bhabha National Institute), Chennai, India

Alexis de Colnet (11)
Algorithms and Complexity Group, TU Wien, Austria

Bruno Dutertre (29)
Amazon Web Services, Santa Clara, CA, USA

Daniel Faber (12)
Department of Computer Science, University of Bonn, Germany

Graham Farr (30)
Department of Data Science and AI, Monash University, Melbourne, Australia

Katalin Fazekas (6)
TU Wien, Austria

Johannes K. Fichte (5)
Linköping University, Sweden

Mathias Fleury (6, 9)
University Freiburg, Germany

Dror Fried (13)
Department of Mathematics and Computer Science, The Open University of Israel, Ra'anana, Israel

Nils Froleyks (6)
Johannes Kepler University, Linz, Austria

Vojtěch Havlena (14)
Brno University of Technology, Czech Republic

Markus Hecher (5)
Massachusetts Institute of Technology, Cambridge, MA, USA

Marijn J. H. Heule (29)
Carnegie Mellon University, Pittsburgh, PA, USA;
Amazon Web Services, Seattle, WA, USA

Tim Hoffmann (5)
Friedrich Schiller University Jena, Germany

Lukáš Holík (14, 15)
Brno University of Technology, Czech Republic

Holger H. Hoos (27)
Chair for AI Methodology, RWTH Aachen University, Germany; Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands;
Department of Computer Science, University of British Columbia, Vancouver, Canada

Alexey Ignatiev (16, 30)
Monash University, Melbourne, Australia

Yoichiro Iida (17)
Graduate School of Information Science and Technology, The University of Tokyo, Japan

Mary Inaba (17)
Graduate School of Information Science and Technology, The University of Tokyo, Japan

Markus Iser (18)
Karlsruhe Institute of Technology, Germany

Dmitry Itsykson (19)
Ben-Gurion University of the Negev, Beer-Sheva, Israel; On leave from Steklov Institute of Mathematics at St. Petersburg, Russia

Adalat Jabrayilov (12)
Adesso SE, Dortmund, Germany

Christoph Jabs (18)
HIIT, University of Helsinki, Finland

Christos Karamanos (16)
Monash University, Melbourne, Australia

Kaspar Kasche (5)
Friedrich Schiller University Jena, Germany

Yaroslav A. Kharkov (29)
AWS Quantum Technologies,
New York, NY, USA

Laura Kovács (9)
TU Wien, Austria

Leszek Aleksander Kołodziejczyk (20)
Institute of Mathematics, University of Warsaw,
Poland

Jean-Marie Lagniez (21)
Univ. Artois, CNRS, CRIL, France

Massimo Lauria (7)
Sapienza Università di Roma, Italy

Ondřej Lengál (14)
Brno University of Technology, Czech Republic

Chu-Min Li (8)
MIS, UR 4290, Université de Picardie Jules
Verne, Amiens, France;
Aix Marseille Univ, Université de Toulon, CNRS,
LIS, Marseille, France

Meena Mahajan (10)
The Institute of Mathematical Sciences (A CI of
Homi Bhabha National Institute), Chennai,
India

Pierre Marquis (21)
Univ. Artois, CNRS, CRIL, IUF, France

Sasank Mouli (10)
Indian Institute of Technology Indore, India

Petra Mutzel (12)
Department of Computer Science, University of
Bonn, Germany

Alexander Nadel (13)
Intel Corporation, Israel and Faculty of Data
and Decision Sciences, Technion, Haifa, Israel

Robert Nieuwenhuis (22)
Barcelogic.com, Barcelona, Spain

Albert Oliveras (22)
Technical University of Catalonia,
Barcelona, Spain

Sergei Ovcharov (19)
St. Petersburg State University, Russia

Tomáš Peitl (31)
Algorithms and Complexity Group,
TU Wien, Austria

Gaurav Rattan (4)
University of Twente, Enschede,
The Netherlands

Franz-Xaver Reichl (23)
TU Wien, Austria

Enric Rodríguez-Carbonell (22)
Technical University of Catalonia,
Barcelona, Spain

Christoph Scholl (24)
Department of Computer Science,
University of Freiburg, Germany

Dominik Schreiber (25)
Karlsruhe Institute of Technology, Germany

Roberto Sebastiani (13)
DISI, University of Trento, Italy

Martina Seidl (1)
Johannes Kepler University Linz, Austria

Tobias Seufert (24)
Department of Computer Science,
University of Freiburg, Germany

Irfansha Shaik (26)
Department of Computer Science, Aarhus
University, Denmark;
Kvantify Aps, Copenhagen S, Denmark

Yogev Shalmon (13)
Intel Corporation, Israel and The Open
University of Israel, Ra’anana, Israel

Hadar Shavit (27)
Chair for AI Methodology,
RWTH Aachen University, Germany

Yunong Shi (29)
AWS Quantum Technologies,
New York, NY, USA

Fabian Siegwolf (24)
Department of Computer Science,
University of Freiburg, Germany

Friedrich Slivovsky (23, 28)
University of Liverpool, UK

Tomohiro Sonobe (17)
National Institute of Informatics, Tokyo, Japan

Peter J. Stuckey (30)
Department of Data Science and AI, Monash University, Melbourne, Australia;
Australian Research Council OPTIMA ITTC, Melbourne, Australia

Stefan Szeider (23, 31)
TU Wien, Austria

Juraj Síč (14)
Brno University of Technology, Czech Republic

Zi Li Tan (16)
Monash University, Melbourne, Australia

Neil Thapen (20)
Institute of Mathematics, Czech Academy of Sciences, Prague, Czech Republic

Cesare Tinelli (2)
The University of Iowa, Iowa City, IA, USA

Jaco van de Pol (26)
Department of Computer Science, Aarhus University, Denmark

Moshe Y. Vardi (3)
Rice University, Houston, TX, USA

Pavol Vargovčík (15)
Faculty of Information Technology, Brno University of Technology, Czech Republic

Jiong Yang (29)
National University of Singapore, Singapore

Furong Ye (8)
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China;
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Jinqiang Yu (30)
Department of Data Science and AI, Monash University, Melbourne, Australia;
Australian Research Council OPTIMA ITTC, Melbourne, Australia

Tianwei Zhang (31)
Algorithms and Complexity Group, TU Wien, Austria

Rui Zhao (22)
Technical University of Catalonia, Barcelona, Spain

Models and Counter-Models of Quantified Boolean Formulas

Martina Seidl
Johannes Kepler University Linz, Austria

Abstract

Because of the duality of universal and existential quantification, quantified Boolean formulas (QBF), the extension of propositional logic with quantifiers over the Boolean variables, have not only solutions in terms of models for true formulas like in SAT. Also false QBFs have solutions in terms of counter-models. Both models and counter-models can be represented as certain binary trees or as sets of Boolean functions reflecting the dependencies among the variables of a formula. Such solutions encode the answers to application problems for which QBF solvers are employed like the plan for a planning problem or the error trace of a verification problem. Therefore, models and counter-models are at the core of theory and practice of QBF solving. In this invited talk, we survey approaches that deal with models and counter-models of QBFs and identify some open challenges.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Quantified Boolean Formula, Solution Extraction, Solution Counting

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.1

Category Invited Talk

Funding Parts of this work were supported by the LIT AI Lab founded by the State of Upper Austria.

1 Overview

The evaluation of a *quantified Boolean formula* (QBF) [8] is often seen as a two-player game between a universal and an existential player: given a QBF $\Phi = \forall X_1 \exists X_2 \ldots \exists X_n.\phi$ where X_i are disjoint sets of variables and ϕ is a propositional formula over these variables, the task is to decide if the QBF is true or false. The existential player aims at satisfying the formula by assigning values to the existentially quantified variables, while the universal player aims at falsifying ϕ by setting the universally quantified variables. The variables need to be assigned in the order as they occur in the prefix. If the formula is true under the chosen assignment, the existential player wins, otherwise the universal player wins. Overall, a QBF is true if and only if there is a winning strategy for the existential player and a QBF is false if and only if there is a winning strategy for the universal player. Nowadays, QBF solvers are applied for many applications [32], and in this context winning strategies play a crucial role. For example, in the context of formal synthesis, a winning strategy for the existential variables encodes the program that is synthesized from a given specification [9], or in the context of a planning problem, the winning strategy encodes the plan [31].

Winning strategies are also often called the *solutions* of a QBF, i.e., they are the *models of true QBFs* and, respectively, the *counter-models of false QBFs*. For a true QBF $\Phi = \forall X_1 \exists X_2 \ldots \exists X_n.\phi$, a model contains the information how to set the values of the existential variables X_{2i} based on the values of the universal variables X_{2j+1} with $0 \leq j < i$ such that ϕ evaluates to true under this assignment. Such a model can be represented either as a binary tree of a certain structure or as a set of Boolean functions, so-called *Skolem functions*. A Skolem function for an existential variable $x \in X_{2i}$ is a Boolean function over

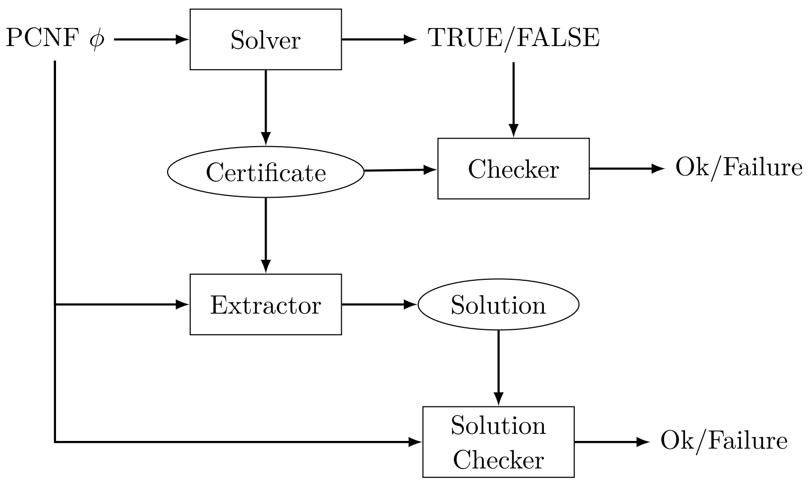


Figure 1 Workflow of solution extraction.

the universal variables X_{2j+1} with $0 \leq j < i$, i.e., the universal variables that precede x in the prefix. If all existential variables are replaced by their Skolem functions, the resulting propositional formula is valid.

Counter-models of false QBFs are defined dually. For a false QBF $\Phi = \forall X_1 \exists X_2 \ldots \exists X_n.\phi$, a counter-model contains the information how to set the values of the universal variables X_{2i+1} based on the values of the existential variables X_{2j} with $0 \leq j \leq i$ such that ϕ evaluates to false under this assignment. Such a counter-model can be represented either as a binary tree of a certain structure or as a set of Boolean functions, so-called *Herbrand functions*. A Herbrand function for an universal variable $x \in X_{2i+1}$ is a Boolean function over the existential variables X_{2j} with $0 \leq j \leq i$, i.e., the existential variables that precede x in the prefix. If all universal variables are replaced by their Herbrand functions, the resulting propositional formula is unsatisfiable.

In QBF research, much emphasis is set on the evaluation of QBFs, i.e., on deciding whether they are true or false. Less effort is set on obtaining models as counter-models despite they are very relevant for practical applications. This is underpinned by the fact that in recent QBF competitions [24, 28] there was no track involving solution extraction, although the benefits of being able to produce winning strategies are manifold. On the one hand models and counter-models can serve as a certificate confirming the correctness of a solving result with the aid of a SAT solver. On the other hand, they encode the solution to the application problem that was translated to a QBF. Furthermore, in the field of proof complexity, counter-models of false QBFs establish a strong tie between theory and practice of QBF solving. In this paper, we take a short tour through the works that explicitly deal with QBF models and counter-models and identify some perspectives for future work.

2 Solutions of Quantified Boolean Formulas

There exist some solvers that generate solutions at runtime. Almost 20 years back, the solver sKizzo was presented [5] which finds BDD-based models for true QBFs. Almost at the same time, the solvers Squolem and EBDDRES were presented that could also generate BDD-based solutions for true formulas [17]. At this time, it was not clear how to combine the generation of solution with clause/cube learning as successfully used in SAT. Later, the abstraction-based solvers CAQE and QuABs were developed that at least in some versions support the extraction of solutions [34]. The solver Cadet [29] for 2QBFs searches for Skolem functions by incrementally adding constraints until these constraints describe a model in terms of Skolem function or until it can be proven that it is not possible to construct a model, i.e., the formula is false. The QBF solver QFun employs machine learning techniques to find short Skolem and Herbrand functions [16].

For a very long time it was unclear how to generate solutions for solvers based on QCDCL, the QBF variant of conflict-driven clause learning [25] which is the major solving paradigm in SAT solving. Independently two approaches were presented that both rely on the fact that QCDCL is based on the Q-resolution calculus [19], the QBF version of resolution. A QCDCL solver like DepQBF [21] can directly emit clause Q-resolution proofs for false formulas and cube Q-resolution proofs for true formulas. For the extraction of models/counter-models, the applications of the QBF-specific existential reduction rule/universal reduction rule has to be taken into account. The approach of Goultiaeva and Van Gelder [12] interactively rewrites a Q-resolution proof as follows. Assume that the considered QBF is false and that the QBF solver produced a clausal Q-resolution refutation. If the outermost variables of the quantifier prefix are existential, then they can be assigned any value and the formula will still evaluate to false. Now some assignment to those variables has to be provided and is applied on the proof, i.e., the variables are assigned. The proof is then simplified resulting in a proof of the QBF under the respective assignment. Now it can be shown that when a universal variable of the new outermost block is eliminated by universal reduction, it occurs only in one polarity. This polarity determines the value of the universal variable. In this way, all values of the outermost universal variables can be directly read off from the proof. Again, the proof is simplified under the assignment resulting in a new proof. Next, the outermost existential variables assigned, resulting again in a proof from which the values of the next outermost universal variables can be read off. This procedure is repeated until all variables are assigned. The other approach by Balabanov and Jiang [1] traverses a Q-resolution proof in reverse topological order and builds Skolem functions from cubes on which existential reduction is applied in the case of true formulas and it builds Herbrand functions from clauses on which universal reduction is applied. In this way, Boolean functions are generated which are typically represented as And-Inverter-Graphs (AIGs). Figure 1 shows the workflow of solution generation after the actual solving: first, the QBF solver decides whether the QBF is true or false and produces a proof. This proof can be efficiently checked by an independent checker to validate the solving result. Furthermore, the proof is then analyzed and the Skolem/Herbrand functions are extracted. By replacing variables by their functions, a SAT solver can then be used to (1) check the correctness of the Skolem/Herbrand functions and (2) to confirm the solving result again. The complete tool chain is implemented in the QBFCert framework [26]. Solution extraction from proofs has been considered for other proof systems than the basic version of Q-resolution like QU-resolution and long-distance resolution [3, 2], for the expansion-based proof system $\forall Exp+Res$ and its extensions [7, 13] as well as for the QRAT proof system [15] mainly used for preprocessing. In [11], we presented an approach to combine the partial solutions obtained from preprocessing with solutions obtained from complete solvers. In practical QBF solving, the extraction of solution is also useful for debugging QBF encodings as suggested in [30].

When dealing with models and counter-models of QBFs as first-class objects, it also becomes relevant to ask for symmetries [18] leading to the distinction of syntactic and semantic symmetries as well as to ask for the overall number of solutions. First approaches to solution counting for true and false QBFs have been presented. First, only assignments to variables of the outermost quantifier block were considered [33, 4] using an enumerative approach, but later this work was lifted to variables at the second quantifier level [27]. Most recently, an efficient recursive approach has been introduced that computes the full model count of true QBFs [10].

The impact of extracting winning strategies is not only practically motivated, but it is also motivated by important results in the field of proof complexity (see [6] for a survey by Beyersdorff). Beyersdorff identifies strategy extraction as a distinctive feature of QBF proof systems for which no propositional analogue exists and states that most QBF lower-bound techniques employ strategy extraction.

3 Some Open Challenges

While there has been made considerable progress in many theoretical and practical aspects of QBF solving, there are still many open challenges that need to be addressed in the future. In the following, three of such challenges are shortly discussed.

Obtaining Solutions for True Formulas

Although solution extraction and generation should be dual for true and false formulas in theory, in practice there is a gap because of QBFs are usually provided in prenex conjunctive normal form (PCNF). The PCNF representation yields several advantages because of the easier implementation and because some techniques work only for clauses, but not for arbitrary formulas. At the same time, the PCNF representation also introduces a bias, because for true formulas having the formula in prenex disjunctive normal form (PDNF) would be preferable. Currently, the PDNF representation has to be constructed during the solving leading to large initial cubes or it involves an expensive transformation. To overcome this problem, it might be preferable to focus on the original structure of a formula and not to flatten it to an equivalent PCNF.

Solutions from Different Solving Paradigms

Over the last years, several different QBF solving paradigm have been shown to be orthogonal in their strength. Therefore, their integration is often beneficial either in terms of distributed portfolio solvers [14] or via well defined interfaces as suggested in [22, 20]. However, when combining multiple approaches, the extraction of solutions becomes challenging, because it is not clear how to define the interfaces at the solution level. We presented some work going in this direction in [11].

Solutions from Parallel and Distributed QBF Solving

Current work shows the potential of exploiting modern distributed and parallel hardware resources for QBF solving [23]. QBFs can be easily split into smaller subproblems that can be handled individually. In the case of SAT, also the solutions can be handled individually: if one of the subproblems is satisfiable, then its solution can be directly extended to the solution of the overall problem. If all subproblems are unsatisfiable, then there exists no solution. For QBFs, the situation is more complicated because the Skolem and Herbrand functions need to be assembled.

References

1 Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Formal Methods Syst. Des.*, 41(1):45–65, 2012. doi:10.1007/S10703-012-0152-6.

2 Valeriy Balabanov, Jie-Hong Roland Jiang, Mikolas Janota, and Magdalena Widl. Efficient extraction of QBF (counter)models from long-distance resolution proofs. In Blai Bonet and Sven Koenig, editors, *Proc. of the 29th AAAI Conference on Artificial Intelligence*, pages 3694–3701. AAAI Press, 2015.

3 Valeriy Balabanov, Magdalena Widl, and Jie-Hong R. Jiang. QBF resolution systems and their proof complexities. In Carsten Sinz and Uwe Egly, editors, *Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2014)*, volume 8561 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2014.

4 Bernd Becker, Rüdiger Ehlers, Matthew Lewis, and Paolo Marin. ALLQBF Solving by Computational Learning. In *Proc. of the 10th Int. Conf. on Automated Technology for Verification and Analysis (ATVA)*, volume 7561 of *LNCS*, pages 370–384. Springer, 2012.

5 Marco Benedetti. skizzo: A suite to evaluate and certify qbfs. In Robert Nieuwenhuis, editor, *Proc. of the 20th Int. Conf. on Automated Deduction (CADE-20)*, volume 3632 of *Lecture Notes in Computer Science*, pages 369–376. Springer, 2005.

6 Olaf Beyersdorff. *Proof Complexity of Quantified Boolean Logic – A Survey*, chapter Chapter 15, pages 397–440. World Scientific, 2023.

7 Olaf Beyersdorff, Leroy Chew, and Mikolas Janota. On unification of QBF resolution-based calculi. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Proc. of the 39th Int. Symposium on Mathematical Foundations of Computer Science (MFCS 2014)*, volume 8635 of *Lecture Notes in Computer Science*, pages 81–93. Springer, 2014.

8 Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1177–1221. IOS Press, 2021.

9 Roderick Bloem, Robert Könighofer, and Martina Seidl. Sat-based synthesis methods for safety specs. In Kenneth L. McMillan and Xavier Rival, editors, *Proc. of the 15th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014)*, volume 8318 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014.

10 Florent Capelli, Jean-Marie Lagniez, Andreas Plank, and Martina Seidl. A top-down tree model counter for quantified boolean formulas. In *Accepted for IJCAI 2024*, 2024.

11 Katalin Fazekas, Marijn J. H. Heule, Martina Seidl, and Armin Biere. Skolem function continuation for quantified boolean formulas. In Sebastian Gabmeyer and Einar Broch Johnsen, editors, *Proc. of the 11th Int. Conf. on Tests and Proofs (TAP 2017)*, volume 10375 of *Lecture Notes in Computer Science*, pages 129–138. Springer, 2017.

12 Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. A uniform approach for generating proofs and strategies for both true and false QBF formulas. In Toby Walsh, editor, *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI 2011)*, pages 546–553. IJCAI/AAAI, 2011.

13 Vedad Hadzic, Roderick Bloem, Ankit Shukla, and Martina Seidl. Ferpmodels: A certification framework for expansion-based QBF solving. In Bruno Buchberger, Mircea Marin, Viorel Negru, and Daniela Zaharie, editors, *Proc. of the 24th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2022)*, pages 80–83. IEEE, 2022.

14 Maximilian Heisinger, Martina Seidl, and Armin Biere. Paraqooba: A fast and flexible framework for parallel and distributed QBF solving. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Proc. of the 29th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023)*, volume 13993 of *Lecture Notes in Computer Science*, pages 426–447. Springer, 2023.

15 Marijn J. H. Heule, Martina Seidl, and Armin Biere. Solution validation and extraction for QBF preprocessing. *J. Autom. Reason.*, 58(1):97–125, 2017.

16 Mikolás Janota. Towards generalization in QBF solving via machine learning. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proc. of the 32nd AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, pages 6607–6614. AAAI Press, 2018.

17 Toni Jussila, Armin Biere, Carsten Sinz, Daniel Kröning, and Christoph M. Wintersteiger. A first step towards a unified proof checker for QBF. In João Marques-Silva and Karem A. Sakallah, editors, *Proc. of the 10th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2007.

18 Manuel Kauers and Martina Seidl. Symmetries of quantified boolean formulas. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Proc. of the 21st Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2018)*, volume 10929 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2018.

19 Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.

20 Florian Lonsing, Fahiem Bacchus, Armin Biere, Uwe Egly, and Martina Seidl. Enhancing search-based QBF solving by dynamic blocked clause elimination. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Proc. of the 20th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*, volume 9450 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2015.

21 Florian Lonsing and Uwe Egly. Depqbf 6.0: A search-based QBF solver beyond traditional QCDCL. In Leonardo de Moura, editor, *Proc. of the 26th Int. Conf. on Automated Deduction (CADE 26)*, volume 10395 of *Lecture Notes in Computer Science*, pages 371–384. Springer, 2017.

22 Florian Lonsing, Uwe Egly, and Martina Seidl. Q-resolution with generalized axioms. In Nadia Creignou and Daniel Le Berre, editors, *Proc. of the 19th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *Lecture Notes in Computer Science*, pages 435–452. Springer, 2016.

23 Florian Lonsing and Martina Seidl. Parallel solving of quantified boolean formulas. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 101–139. Springer, 2018.

24 Florian Lonsing, Martina Seidl, and Allen Van Gelder. The QBF gallery: Behind the scenes. *Artif. Intell.*, 237:92–114, 2016.

25 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021.

26 Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF – (tool presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *Proc. of the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *Lecture Notes in Computer Science*, pages 430–435. Springer, 2012.

27 Andreas Plank, Sibylle Möhle, and Martina Seidl. Enumerative level-2 solution counting for quantified boolean formulas (short paper). In *Proc. of the 29th Int. Conf. on Principles and Practice of Constraint (CP)*, volume 280 of *LIPIcs*, pages 49:1–49:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

28 Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17). *Artif. Intell.*, 274:224–248, 2019.

29 Markus N. Rabe and Sanjit A. Seshia. Incremental determinization. In Nadia Creignou and Daniel Le Berre, editors, *Proc. of the 19th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT 2016)*, volume 9710 of *Lecture Notes in Computer Science*, pages 375–392. Springer, 2016.

30 Irfansha Shaik, Maximilian Heisinger, Martina Seidl, and Jaco van de Pol. Validation of QBF encodings with winning strategies. In Meena Mahajan and Friedrich Slivovsky, editors, *Proc. of the 26th Int. Conf. on Theory and Applications of Satisfiability (SAT 2023)*, volume 271 of *LIPIcs*, pages 24:1–24:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

31 Irfansha Shaik and Jaco van de Pol. Classical planning as QBF without grounding. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proc. of the 32nd Int. Conf. on Automated Planning and Scheduling (ICAPS 2022)*, pages 329–337. AAAI Press, 2022.

32 Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified boolean formulas. In *Proc. of the 31st IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI 2019)*, pages 78–84. IEEE, 2019.

33 Ankit Shukla, Sibylle Möhle, Manuel Kauers, and Martina Seidl. Outercount: A first-level solution-counter for quantified boolean formulas. In *Proc. of the 15th Int. Conf on Intelligent Computer Mathematics (CICM 2022)*, volume 13467 of *Lecture Notes in Computer Science*, pages 272–284. Springer, 2022.

34 Leander Tentrup. CAQE and quabs: Abstraction based QBF solvers. *J. Satisf. Boolean Model. Comput.*, 11(1):155–210, 2019.

Scalable Proof Production and Checking in SMT

Cesare Tinelli
The University of Iowa, Iowa City, IA, USA

Abstract

Solvers for Satisfiability Modulo Theories (SMT) have become crucial components in safety- or mission-critical formal methods applications, in particular model checking, verification, and security analysis. Since state-of-the-art SMT solvers are large and complex systems, they are prohibitively difficult to prove correct. Hence, proof production is essential as a way to demonstrate instead the correctness of their responses, making those responses amenable to independent verification. Historically, the main challenges for proof production in SMT have been solver performance and proof coverage, often leading to the disabling of many sophisticated solving techniques when running in proof-production mode, or to coarse-grained, and harder to check, proofs.

The first part of this talk presents a flexible proof-production architecture designed to handle the complexity of versatile, industrial-strength SMT solvers, and discusses how it has been leveraged to produce detailed proofs, even for sophisticated reasoning components. The architecture, implemented in the state-of-the-art SMT solver cvc5, allows proofs to be produced modularly, as needed, and with various safeguards for correctness. The architecture supports the generation of textual proof certificates in different formats, for offline proof checking by external tools, as well as a rich API, which is useful for online integration of the SMT solver into other reasoning tools such as, for instance, skeptical proof assistants. Extensive experimental evaluations with both SMT-LIB benchmarks and benchmarks provided by industrial partners have shown that the new architecture results in greater proof coverage than previous approaches, imposes a small runtime overhead, and supports fine-grained proofs in the great majority of cases.

The second part of the talk gives an overview of a new generic language for expressing SMT proof certificates that builds on almost two decades of work and experience in proof generation and checking in SMT and combines the benefits of several previous efforts on the topic. While developed to express cvc5's proof certificates, the language is meant to be useful to other SMT solvers as well. It is in fact a logical framework, based on the syntax and semantics of the upcoming Version 3 of the SMT-LIB standard, that can be customized, as in the case of cvc5, with the specific proof system used by the solver through the definition of new symbols, binders and proof rules. In addition, it features an intuitive syntax for representing natural-deduction-style proofs and the ability to integrate other proof formats (such as, for instance, those currently used by SAT solvers) via the use of oracles. The talk discusses an initial evaluation of the proof language, obtained with a companion checker for it and an instantiation to cvc5's proof system. The evaluation shows the viability of high-performance, fine-grained proof production and checking for SMT.

The talk concludes with a brief overview of future work and new potential applications enabled by scalable proof certificate production and checking.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Theory of computation → Proof theory; Theory of computation → Type theory

Keywords and phrases Satisfiability Modulo Theories, Proof generation and certification

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.2

Category Invited Talk

Supplementary Material *Software (Source code of cvc5 SMT solver)*: `https://github.com/cvc5/cvc5`, archived at `swh:1:dir:2f245f4f16b574408f7861a2d7a4a763e47e5a7f`

Funding The work described in this talk was funded in part by Amazon Web Services and by DARPA award FA8750-24-2-1001.

Acknowledgements This work has involved many collaborators over the course of almost 4 years. Significant contributors are (in alphabetical order): Leni Aniva, Haniel Barbosa, Clark Barrett, Tomaz Gomes, Harun Khan, Gereon Kremer, Hanna Lachnitt, Abdalrhman Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Hans-Jörg Schurr, Arjun Viswanathan, and Yoni Zohar.

Logical Algorithmics: From Relational Queries to Boolean Reasoning

Moshe Y. Vardi
Rice University, Houston, TX, USA

Abstract

The standard approach to algorithm development is to focus on a specific problem and develop for it a specific algorithm. Codd's introduction of the relational model in 1970 included two fundamental ideas: (1) relations provide a universal data representation formalism, and (2) relational databases can be queried using first-order logic. Realizing these ideas required the development of a meta-algorithm, which takes a declarative query and executes it with respect to a database. In this talk, I will describe this approach, which I call Logical Algorithmics, in detail, and trace a decades-long path from the comoutational complexity theory of relational queries to recent tools for Boolean reasoning.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography; Theory of computation → Design and analysis of algorithms

Keywords and phrases Logic, Algorithms

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.3

Category Invited Talk

Funding Work supported in part by NSF grants IIS-1527668, CCF-1704883, IIS-1830549, CNS-2016656, DoD MURI grant N00014-20-1-2787, and an award from the Maryland Procurement Office.

References

1 J.M. Dudek, Vu Phan, and M.Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *The 34th AAAI Conference on Artificial Intelligence*, pages 1468–1476. AAAI Press, 2020.
2 P.G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.
3 G. Pan and M.Y. Vardi. Symbolic techniques in satisfiability solving. *J. Autom. Reason.*, 35(1-3):25–50, 2005.
4 V.H.N. Phan and M.Y. Vardi. DPO: dynamic-programming optimization on hybrid constraints. *CoRR*, abs/2205.08632, 2022. `arXiv:2205.08632`.
5 M.Y. Vardi. The complexity of relational query languages. In *Proc. 14th ACM Symposium on Theory of Computing*, pages 137–146. ACM, 1982.
6 M.Y. Vardi. On the complexity of bounded-variable queries. In *Proc. 14th ACM Symp. on Principles of Database Systems*, pages 266–76, 1995.

Satsuma: Structure-Based Symmetry Breaking in SAT

Markus Anders
TU Darmstadt, Germany

Sofia Brenner
TU Darmstadt, Germany

Gaurav Rattan
University of Twente, Enschede, The Netherlands

Abstract

Symmetry reduction is crucial for solving many interesting SAT instances in practice. Numerous approaches have been proposed, which try to strike a balance between symmetry reduction and computational overhead. Arguably the most readily applicable method is the computation of static symmetry breaking constraints: a constraint restricting the search-space to non-symmetrical solutions is added to a given SAT instance. A distinct advantage of static symmetry breaking is that the SAT solver itself is not modified. A disadvantage is that the strength of symmetry reduction is usually limited. In order to boost symmetry reduction, the state-of-the-art tool BreakID [Devriendt et. al] pioneered the identification and tailored breaking of a particular substructure of symmetries, the so-called row interchangeability groups.

In this paper, we propose a new symmetry breaking tool called satsuma. The core principle of our tool is to exploit more diverse but frequently occurring symmetry structures. This is enabled by new practical detection algorithms for row interchangeability, row-column symmetry, Johnson symmetry, and various combinations. Based on the resulting structural description, we then produce symmetry breaking constraints. We compare this new approach to BreakID on a range of instance families exhibiting symmetry. Our benchmarks suggest improved symmetry reduction in the presence of Johnson symmetry and comparable performance in the presence of row-column symmetry. Moreover, our implementation runs significantly faster, even though it identifies more diverse structures.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases symmetry breaking, boolean satisfiability, graph isomorphism

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.4

Related Version *Preprint*: `https://arxiv.org/abs/2406.13557`

Supplementary Material *Software (source code)*: `https://github.com/markusa4/satsuma` [2]
archived at `swh:1:dir:134ea7952a25d92ba2addf7ff25aa18183550735`

Funding The research leading to these results has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EngageS: grant agreement No. 820148). Sofia Brenner additionally received funding from the German Research Foundation DFG (SFB-TRR 195 "Symbolic Tools in Mathematics and their Application").

1 Introduction

Symmetries are present in many interesting SAT instances, ranging from hard combinatorial problems to circuit design. Making use of symmetry is paramount in order to efficiently solve many of these instances. Practical approaches for symmetry reduction must always strike a balance between the computational overhead incurred and the strength of the symmetry reduction. Two decades of research have led to many approaches to tackle this problem [17, 1, 21, 36, 22, 20, 34, 26, 30]. At one end of the spectrum, isomorph-free generation techniques [26, 30] apply sophisticated algorithms in conjunction with the solver, such that a solver only explores asymmetric branches of the search. While these techniques are successful in solving hard combinatorial instances (e.g., [29]), this comes at the price of substantial overhead: both in terms of computational cost as well as interfering with the other strategies employed by solvers. Hence, one must be sure that the symmetry reduction is worth the additional overhead. It therefore seems impractical to turn these techniques "on-by-default".

Arguably at the other end are tools producing *static symmetry breaking constraints* [17, 1, 21]. These tools add additional clauses and variables to a given instance, with the aim of reducing the number of symmetric branches explored by the solver. While the symmetry reduction is usually not as strong as for dynamic techniques, such constraints can be computed comparatively cheaply. More importantly, a distinct advantage of static symmetry breaking constraints is that the SAT solver itself is not modified, and hence there is a complete separation of concerns. State-of-the-art static symmetry breaking tools are successfully applied as an "on-by-default" technique [14, 21]. Static symmetry breaking is not only used in SAT, but also in various other areas of constraint programming [25, 35, 19, 8].

In order to improve symmetry reduction, a rather recent development in static symmetry breaking is to detect and make use of so-called *row interchangeability* subgroups [21, 35]. In SAT, this feature was introduced by the state-of-the-art symmetry breaking tool BreakID [21], but it is also used in symmetry breaking in mixed integer programming (MIP) [35]. Row interchangeability groups stem from a natural modeling of the variables as a matrix in which all *rows* are interchangeable by a symmetry. The idea is to first identify these row interchangeability groups, and then produce tailored symmetry breaking constraints. The current generation of tools identifies row interchangeability by hoping for and exploiting a particular structure in the generators of the symmetry group. However, the method is not guaranteed to work and sometimes incurs significant overhead [5]. Despite this, the gain in symmetry reduction seems to be worth the trade-off [21].

In the realm of constraint programming, symmetry breaking constraints for more structures have been considered: for example, *row-column symmetry* [23] is a natural extension of row interchangeability, where both the rows and columns are interchangeable. These symmetries are common in combinatorics, scheduling, or assignment problems [24, 23], such as the well-known pigeonhole principle. While traditional complete symmetry breaking constraints are unlikely to be efficiently computable for these structures [23], different practical constraints are well-studied in the literature for *manual* breaking of symmetries [23, 28]. Another area in which symmetry breaking has been studied in detail is graph generation [15, 16, 30]. A typical problem in this area is to determine the existence of a graph with a specific property. Symmetries in these problems often simply correspond to *isomorphic graphs*. Even though this is rarely mentioned explicitly, such symmetries can be described by so-called *Johnson groups* [9, 10].

In *automated* symmetry breaking, making use of such results requires us to *identify* the appropriate structures first. However, generalizing the existing identification strategies of contemporary tools to more elaborate structures seems elusive.

1.1 Contribution

We present a new algorithm for the generation of symmetry breaking constraints, and a prototype implementation called satsuma. Our goal is to explore whether the approach of "identifying and exploiting specific group structures" can be pushed further.

Techniques. We place the identification of specific symmetry groups at the very heart of satsuma. The approach is enabled by our main contribution, a new class of practical detection algorithms. In particular, we provide algorithms identifying row interchangeability

(Section 3.1), row-column symmetry (Section 3.2), and Johnson symmetry (Section 3.3). Furthermore, we detect certain *combinations* of the above groups, as well as groups which are *similar* to the above groups, building essentially a *structural description* of the group. Symmetry breaking constraints are then chosen based on the type of detected structure: for each detected structure, we determine a set of carefully chosen symmetries, for which conventional symmetry breaking constraints are produced.

Our detection algorithms are all based on the highly efficient *individualization-refinement* framework, as is commonly used in practical graph isomorphism algorithms [33]. Our detection algorithms are all heuristics, in that identification of a particular group cannot be guaranteed. However, the success of the heuristics provably depends only on a well-studied graph property (see Section 2).

These algorithms can be applied *without* computing the symmetries of the formula first: they are purely graph-based. We exploit this by first running our tailored detection algorithms, and then only apply general-purpose symmetry detection on parts not yet identified. In order to handle this remainder, SATSUMA reimplements parts of BREAKID. Essentially, our new approach acts as a preprocessor for existing techniques.

Benchmarks. We compare SATSUMA and BREAKID on a range of well-established SAT instance families exhibiting symmetry. In our benchmarks, we observe that our new structure-based implementation

1. leads to improved SAT solver performance on instances with Johnson symmetries,
2. comparable SAT solver performance on instances exhibiting predominantly row interchangeability or row-column symmetry,
3. and incurs less computational overhead on all tested benchmark families (we observe better asymptotic scaling of SATSUMA on some benchmark families).

When SATSUMA detects a structure, our new approach seems to be a win-win: it yields lower computational overhead, and the resulting speed-up for SAT solvers is comparable or better.

2 Preliminaries

2.1 Satisfiability and Symmetry

SAT. In this paper, a SAT formula F in *conjunctive normal form* (CNF) is denoted with $F = \{\{l_{1,1}, \cdots, l_{1,k_1}\}, \cdots, \{l_{m,1}, \cdots, l_{m,k_m}\}\}$. Each element $C \in F$ is called a *clause*, whereas a clause itself consists of a set of *literals*. A literal is either a variable v or its negation $\neg v$. We write $\mathrm{Var}(F) \coloneqq \{v_1, \ldots, v_n\}$ for the set of *variables* of F and use $\mathrm{Lit}(F)$ for its literals.

A symmetry, or *automorphism*, of F is a permutation $\varphi\colon \mathrm{Lit}(F) \to \mathrm{Lit}(F)$ satisfying the following properties. First, it maps F to itself, i.e., $F^\varphi = F$, where F^φ means applying φ element-wise to the literals in each clause. Second, for all $l \in \mathrm{Lit}(F)$ it holds that $\neg\varphi(l) = \varphi(\neg l)$. We define the *support* of φ as $\mathrm{supp}(\varphi) = \{l \in \mathrm{Lit}(F)\colon l^\varphi \neq l\}$, i.e., the set of all literals moved by φ. The set of all symmetries of F is $\mathrm{Aut}(F)$. We can efficiently *test* if a permutation φ is an automorphism of F: for each clause C, we check whether $C^\varphi \in F$ holds.

An *assignment* of F is a function $\theta\colon \mathrm{Var}(F) \to \{0, 1\}$. We define the evaluation of F under θ in the usual way, i.e., either $F[\theta] = 1$ or $F[\theta] = 0$ holds. A formula F is satisfiable if there exists an assignment θ with $F[\theta] = 1$, and unsatisfiable otherwise. Given an assignment θ of F and an automorphism $\varphi \in \mathrm{Aut}(F)$, we define $\theta^\varphi(v) \coloneqq \theta(v')$ if $\varphi(v) = v'$ for $v' \in \mathrm{Var}(F)$ and $\theta^\varphi(v) \coloneqq \neg\theta(v')$ if $\varphi(v) = \neg v'$ for $v' \in \mathrm{Var}(F)$, where naturally $\neg 0 = 1$ and $\neg 1 = 0$. It follows readily that for $\varphi \in \mathrm{Aut}(F)$, we have $F[\theta] = F^\varphi[\theta^\varphi] = F[\theta^\varphi]$.

Symmetry Breaking Constraints. All symmetry breaking constraints in this paper are *lex-leader* constraints. Let $\prec$ denote a total order of $\mathrm{Var}(F)$. We order an assignment θ according to $\prec$, yielding a $\{0,1\}$-string. We can then order assignments θ, θ' of F lexicographically by comparing their corresponding strings, denoted by $\prec_{\mathrm{lex}}$. Given an automorphism φ of F, it suffices to evaluate F on those assignments θ for which $\theta^\varphi \preceq_{\mathrm{lex}} \theta$ holds, since $F[\theta^\varphi] = F[\theta]$. In particular, we may add a *lex-leader constraint* $\mathrm{LL}_\varphi^\prec$ to F, which ensures that $\theta^\varphi \preceq_{\mathrm{lex}} \theta$ holds. It is easy to see that F is satisfiable, if and only if $F \bigwedge_{\varphi \in \mathrm{Aut}(F)} \mathrm{LL}_\varphi^\prec$ is satisfiable [37]. Lex-leader constraints can be efficiently encoded as a CNF formula, and different encodings have been studied in detail [1, 21]. The practical encoding we use is reverse-engineered from BreakID, and is described in [21]. Having detected structures of symmetries, satsuma attempts to determine a favorable variable order and a set of automorphisms for which lex-leader constraints are constructed (see Section 4).

2.2 Graphs and Symmetry

Graphs. An undirected graph $G = (V, E)$ consists of a vertex set V and an edge relation $E \subseteq \binom{V}{2}$. We refer to the set of vertices of G as $V(G)$, and to the set of edges as $E(G)$. A *vertex coloring* of G is a mapping $\pi : V(G) \to [k]$ to colors in $[k]$ for some $k \in \mathbb{N}$. We call (G, π) a vertex-colored graph. The *color class* of a color c consists of all vertices of G with color c. The color classes form a partition of $V(G)$, the *color partition* corresponding to π.

A bijection $\varphi : V(G) \to V(G)$ is called an *automorphism* of (G, π), whenever $(G, \pi)^\varphi = (G^\varphi, \pi^\varphi) = (G, \pi)$ holds. Here, G^φ denotes the graph with vertex set $V(G)$ and edges $\{u^\varphi, v^\varphi\}$ whenever $\{u, v\}$ is an edge of G (where v^φ simply denotes the image of v under φ). The coloring π^φ is given by $\pi^\varphi(v) = \pi(v^\varphi)$ for every $v \in V(G)$. The set of all automorphisms of (G, π) is denoted by $\mathrm{Aut}(G, \pi)$.

For a given CNF formula F, we define the *model graph* $G(F) = (G, \pi)$ as follows. The vertex set consists of the literals and clauses of F. There are edges connecting the literals of a common variable to each other. Clauses are connected to the literals they contain. Formally, let $E := \{\{v, \neg v\} : v \in \mathrm{Var}(F)\} \cup \{\{C, l\} : l \in C, C \in F\}$. Define a coloring π by setting $\pi(l) := 0$ for all literals $l \in \mathrm{Lit}(F)$ and $\pi(C) := 1$ for all clauses $C \in F$. It is well-known that the automorphisms of $G(F)$ restricted to $\mathrm{Lit}(F)$ are precisely the automorphisms of F [37].

Permutation Groups. We recall some notions of permutation group theory. A detailed account can be found in [38]. Let Ω be a nonempty finite set. Let $\mathrm{Sym}(\Omega)$ denote the *symmetric group* on Ω, i.e., the group of permutations of Ω, and set $\mathrm{Sym}(n) := \mathrm{Sym}([n])$. A *permutation group* is a subgroup Γ of $\mathrm{Sym}(\Omega)$, denoted by $\Gamma \leq \mathrm{Sym}(\Omega)$. We also say that Γ *acts on* Ω. For $g \in \Gamma$ and $\omega \in \Omega$, we write ω^g for the image of ω under g and $\omega^\Gamma = \{\omega^g : g \in \Gamma\}$ for the *orbit* of ω under Γ. In other words, ω^Γ consist of all points in Ω that can be reached from ω by applying elements of Γ. The partition of Ω into the orbits of Γ is called the *orbit partition.* For $\omega \in \Omega$, let $\Gamma_\omega := \{g \in \Gamma : \omega^g = \omega\}$ denote the *stabilizer* of ω in Γ. In other words, Γ_ω consists of those elements in Γ that map ω to itself. The *direct product* of permutation groups Γ_1 and Γ_2 which act on domains Ω_1 and Ω_2, respectively, is the Cartesian product $\Gamma_1 \times \Gamma_2$, endowed with a component-wise multiplication. It naturally acts component-wise on $\Omega_1 \times \Omega_2$.

Individualization-Refinement. A central ingredient in our algorithms is the so-called *individualization-refinement* (IR) paradigm. The IR paradigm is the central technique in all state-of-the-art symmetry detection algorithms [33, 18, 27, 4], and highly engineered

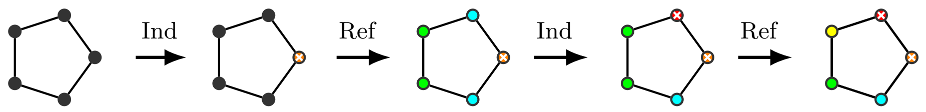


Figure 1 An illustration of the IR process. Individualization steps break symmetries or similarities (nodes marked with a cross are individualized). Refinement steps propagate this information.

implementations are available. The paradigm mainly consists of the *individualization* technique, paired with the so-called *color refinement algorithm.* In this paragraph, we focus on a high-level explanation of the routine. A detailed account can be found in [33].

The central idea of IR (see Figure 1 for an illustration) is the following: given a vertex-colored graph (G, π) and a vertex $v \in V(G)$, the vertex v is *individualized.* Basically this means that it obtains a new color. The routine then proceeds with a so-called *color refinement*: in each step, every vertex of G obtains a new color, based on its former color together with the colors of its neighbors in G. This recoloring procedure is repeated until the corresponding color partition stabilizes. The final coloring π' is then returned. We use the notation $\pi' = \mathrm{IR}((G, \pi), v)$ to denote this process. The call $\mathrm{IR}((G, \pi), v)$ can be computed in time $\mathcal{O}(|E(G)| \log |V(G)|)$ (see [11]).

The coloring π' is a *refinement* of π in the sense that vertices with the same color in π' already had the same color in π. In other words, a color c of π is either preserved in π', or partitioned into several other colors $c_1, \dots, c_n$. For $i \in [n]$, we call the sets $\{u \in V(G) \colon \pi(u) = c,\ \pi'(u) = c_i\}$ the *fragments* of c in π'. The second crucial observation is that vertices in the same orbit under the stabilizer $\mathrm{Aut}(G, \pi)_v$ obtain the same color in π'. However, it is possible that the color partition of π' is *coarser* than the orbit partition in the sense that the vertices of multiple orbits might obtain the same color in π'.

Clearly, this process can be applied inductively to individualize multiple vertices. It is also possible to pass the empty sequence ε to IR, i.e., to run only the color refinement procedure. Arguing as above, the resulting color partition is guaranteed to be at least as coarse as the orbit partition of $\mathrm{Aut}(G, \pi)$ (i.e., the stabilizer of the empty sequence).

The next lemma summarizes the properties of IR to which we refer throughout the paper:

▶ **Lemma 1.** *Given a vertex-colored graph (G, π) and a vertex $v \in V(G)$, the refined coloring $\pi' = \mathrm{IR}((G, \pi), v)$ has the following properties.*

1. *The coloring π' is a* refinement *of π: for $u, w \in V(G)$ with $\pi'(u) = \pi'(w)$, we have $\pi(u) = \pi(w)$.*
2. *The color partition of π' is at least as* coarse *as the orbit partition of $\Gamma = \mathrm{Aut}(G, \pi)_v$: vertices $u, w \in V(G)$ with $\pi'(u) \neq \pi'(w)$ lie in different orbits of Γ, i.e., we have $w \notin u^{\Gamma}$.*
3. *The colors of π' are* isomorphism-invariant*: for every $\varphi \in \mathrm{Sym}(V(G))$, it holds that $\mathrm{IR}((G^{\varphi}, \pi^{\varphi}), v^{\varphi}) = \mathrm{IR}((G, \pi), v)^{\varphi}$. In particular, if $\varphi \in \mathrm{Aut}(G, \pi)$, then $\mathrm{IR}((G, \pi), v)^{\varphi} = \mathrm{IR}((G^{\varphi}, \pi^{\varphi}), v^{\varphi}) = \mathrm{IR}((G, \pi), v^{\varphi})$ holds.*

These properties follow almost immediately from the definition of IR, and we refer to [33] for a treatment of the topic. We also mention that usually, as opposed to the description above, IR is defined for *sequences* of vertices instead of single vertices.

We now recall the notion of Tinhofer graphs [7]. In view of the second part of Lemma 1, these are precisely the graphs for which the two partitions coincide.

▶ **Definition 2** (Tinhofer Graph [7, 40]). *A graph G is called Tinhofer if for all $v \in V(G)$, the orbit partition of $\Gamma := \mathrm{Aut}(G, \pi)_v$ coincides with the color partition of $\pi' := \mathrm{IR}((G, \pi), v)$ and the same applies recursively to the colored graph (G, π') (this corresponds to individualizing multiple vertices of G). Formally, the first property means that for all $u, w \in V(G)$, we have $w \in u^{\Gamma}$ if and only if $\pi'(u) = \pi'(w)$.*

In particular, IR works well on Tinhofer graphs: practical graph isomorphism solvers are guaranteed to not require any backtracking.

2.3 Symmetry Structures in SAT

The idea of our tool is to detect certain symmetry structures that are subsequently exploited. In this section, we describe the main structures detected by the tool. The description of the detection algorithms is the subject of Section 3.

Throughout, let F be a SAT formula. As a first step, consider the *disjoint direct decomposition* of the symmetries $\mathrm{Aut}(F)$: this is a partition $\mathrm{Lit}(F) = L_1 \dot\cup \cdots \dot\cup L_k$ of $\mathrm{Lit}(F)$ for which there exists a decomposition $\mathrm{Aut}(F) = A_1 \times \cdots \times A_k$ into a direct product of subgroups such that, for every $i \in [k]$, the automorphisms in A_i only move the literals in L_i. A disjoint direct decomposition naturally decomposes the symmetry breaking problem, and it suffices to treat each factor separately. In the following, we always refer to the finest such decomposition, which is clearly unique. We call its parts $L_1, \ldots, L_k$ the *disjoint direct factors* of F. Note that every disjoint direct factor is a union of orbits of $\mathrm{Aut}(F)$.

As factors in the disjoint direct decomposition, we detect several variants of three main kinds of symmetries, namely row symmetry, row-column symmetry, and Johnson symmetry. Let us now define these notions in the special context of CNF formulas.

Row Symmetry. *Row interchangeability*, or *row symmetry*, naturally occurs in the context of matrix modeling [24] and is already successfully exploited in automated symmetry breaking. We say that a SAT formula F *exhibits row symmetry* if there exists a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ which can be arranged in a matrix M such that $\mathrm{Aut}(F)|_L$ acts by permuting the rows of M. In addition, we require that every column of M is an orbit of $\mathrm{Aut}(F)$. See Figure 2a for an illustration. The colored boxes illustrate orbits, whereas dashed lines indicate vertices in the same row. The rows can be permuted using symmetry.

We should address a technical difference between the definition above and how BREAKID handles row symmetry: in our definition, a disjoint direct factor should *only* admit the action of the row symmetry group, or a particularly defined extension (see Section 3). BREAKID on the other hand would accept any row symmetry *subgroup* that it detects (see [5] for further discussion). Hence, in practice, it may happen that BREAKID reports row symmetry, when SATSUMA does not. However, SATSUMA may instead identify a larger, more expressive group, such as row-column symmetry, as explained below.

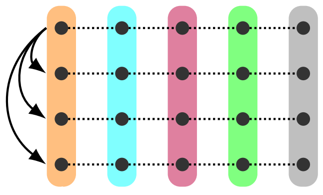
Let us make a general observation regarding negation symmetry.

▶ Remark 3. For an orbit σ of literals under $\mathrm{Aut}(F)$, also the set $\neg\sigma := \{\neg v \colon v \in \sigma\}$ is an orbit of literals. Hence two cases can occur: either we have $\sigma = \neg\sigma$, or the orbits σ and $\neg\sigma$ are disjoint.

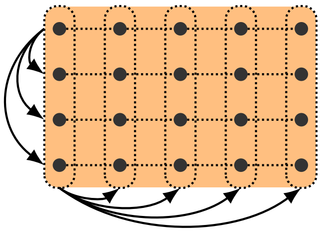
In order to simplify the exposition, we only consider the second scenario in the following.

Row-column symmetry. Row-column symmetries are an extension of row interchangeability. Row-column symmetry naturally occurs whenever both the rows and columns of a matrix of variables are interchangeable. Examples can be found in scheduling, design, and combinatorial problems (see [24]).

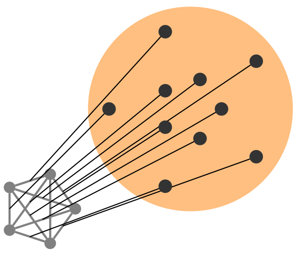
For $m, n \in \mathbb{N}$, the *row-column symmetry* group is $\Gamma := \mathrm{Sym}(n) \times \mathrm{Sym}(m)$, acting componentwise on $[n] \times [m]$. We think of $[n] \times [m]$ as an $n \times m$ matrix M, on which $(\sigma_1, \sigma_2) \in \Gamma$ acts by permuting the n rows according to σ_1 and the m columns according to σ_2.



(a) A row symmetry with 4 rows and 5 columns.

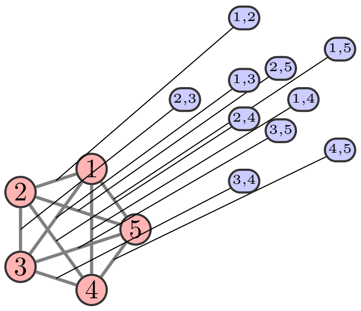


(b) A row-column symmetry with 4 rows and 5 columns.

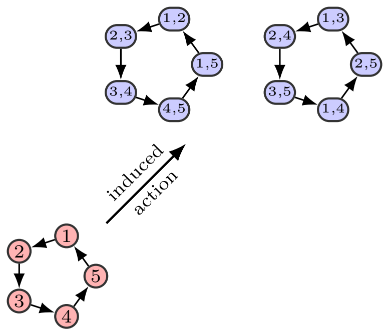


(c) Illustration for the Johnson group $\mathcal{J}_5$. The $\binom{5}{2}$ nodes correspond to edges of a complete graph of size 5.

Figure 2 Various group structures used throughout the paper. Colors indicate orbits of the group.



(a) The domain of $\mathcal{J}_5$ consists of all 2-subsets of the base set $\{1, \ldots, 5\}$.



(b) The action of the 5-cycle on the base set and its induced action on the domain of $\mathcal{J}_5$.

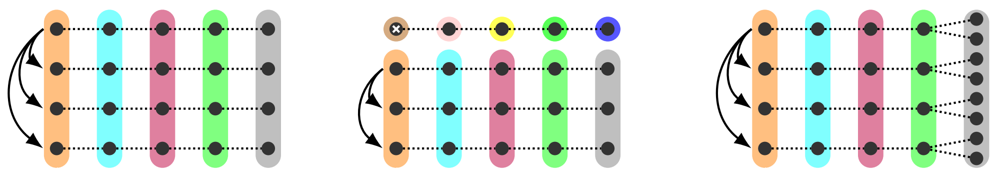
Figure 3 An illustration of the Johnson group $\mathcal{J}_5$.

A SAT formula F *exhibits row-column symmetry* if there exists a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ consisting of an orbit σ of $\mathrm{Aut}(F)$ and its negation $\neg\sigma$ such that the following holds: the literals in σ can be arranged in an $n \times m$-matrix M such that $\mathrm{Aut}(F)|_\sigma$ acts as a row-column symmetry group on M. See Figure 2b for an illustration. Note that the action of $\mathrm{Aut}(F)$ on σ naturally extends to a row-column symmetry action on $\neg\sigma$. For this reason, our algorithm generates the matrix M of the literals in σ and extends this to $\neg\sigma$, see Section 3.2 for details.

Johnson symmetry. Johnson groups are naturally tied to the graph isomorphism problem. Whenever a problem asks for the existence of an undirected graph with a certain property, typically, the underlying symmetries form a Johnson group.

Observe that $\pi \in \mathrm{Sym}(n)$ induces a permutation on the domain $\binom{[n]}{2}$ of 2-subsets of $[n]$, mapping $\{a_1, a_2\}$ to $\{a_1^\pi, a_2^\pi\}$. This way, $\mathrm{Sym}(n)$ becomes a permutation group on a domain of size $\binom{|n|}{2}$, the *Johnson group* $\mathcal{J}_n$. Technically, these groups are specifically Johnson groups of arity 2. The corresponding action is called a *Johnson action*.

We now define Johnson symmetries for SAT formulas. Intuitively, the variables the formula correspond to the "edges" (i.e., sets of two vertices) of a complete graph. There is a symmetric action on the "vertices" of this underlying graph and the variables of the formula ("edges") are permuted accordingly. See Figure 2c and Figure 3 for an illustration. Formally, a SAT formula F *exhibits a Johnson symmetry* if the following holds: there exists



(a) A row symmetry with 4 rows and 5 columns.

(b) Individualizing a vertex of a row identifies the entire row.

(c) A row symmetry on blocks of size 2.

Figure 4 Illustrations of different aspects of row symmetry.

a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ consisting of an orbit σ of $\mathrm{Aut}(F)$ and its negation $\neg\sigma$ such that the literals in σ can be relabeled as $x_{\{i,j\}}$ for all $\{i,j\} \in \binom{[n]}{2}$ and $\mathrm{Aut}(F)|_\sigma$ acts as the Johnson group $\mathcal{J}_n$ (by permuting the index sets). Again, the action of $\mathrm{Aut}(F)$ naturally extends to $\neg\sigma$.

3 Detection Algorithms

We now present our detection algorithms. All algorithms are centered around detecting structure on the model graph $G(F)$ of a given CNF formula F. Recall that $G(F)$ contains a vertex for each literal, so we may use these terms interchangeably. The major design principles of our algorithms are described in the following.

Colors are Orbits. Our algorithms work on the assumption that the model graph $G(F)$ is Tinhofer (see Definition 2). Then we can compute orbits of stabilizers using IR. In particular, the color classes of $\pi = \mathrm{IR}(G(F), \varepsilon)$ *are* then the orbits of $\mathrm{Aut}(G(F))$.

Certified Correctness. The input model graph might not be Tinhofer. However, each algorithm constructs a carefully chosen set of candidate permutations, which suffices to prove the existence of a certain group action. It is then verified that these permutations are automorphisms of the formula F, which ensures correctness. In our implementation, we produce lex-leader constraints only for automorphisms verified on the original formula.

Color-by-color. All of our detection algorithms proceed color-by-color, or orbit-by-orbit: given an orbit, the algorithms stabilize a specific set of points, observing the effect on the given orbit as well as other orbits. If an orbit exhibits a specific group action, then this effect is clearly defined, and a model of the purported structure is made.

3.1 Row Symmetry

We describe an algorithm for row symmetry. First, we define an auxiliary function that transposes two pair-wise disjoint lists of literals of equal length: For $l \in \mathrm{Lit}(F)$, let

$$\mathrm{transpose}_F((l_1 \dots l_k), (l'_1 \dots l'_k))_F(l) \coloneqq \begin{cases} l'_i & \text{if } l = l_i \text{ with } i \in [k] \\ l_i & \text{if } l = l'_i \text{ with } i \in [k] \\ l & \text{otherwise.} \end{cases}$$

Algorithm 1 Detection algorithm for row symmetry.

```
function DetectRowSymmetry
    Input : ➢ formula $F$
            ➢ set $\sigma \subseteq \mathrm{Lit}(F)$ with $|\sigma| \geq 3$
    Output : ≺ matrix with row symmetry including $\sigma$, or $\perp$ if check fails
    $(G, \pi) := G(F)$, $\pi' := \mathrm{IR}((G, \pi), \varepsilon)$;
    // construct a candidate row for each $v \in \sigma$
    foreach $v \in \sigma$ do
        $\pi_v := \mathrm{IR}((G, \pi'), v)$;
        let $\tau$ be a list of literals that are singletons in $\pi_v$ but not in $\pi'$;
        sort literals in $\tau$ according to their color in $\pi_v$;
        $\mathrm{row}[v] := \tau$;
    check that rows are pair-wise disjoint;
    // verify that $M$ exhibits row symmetry
    foreach $i \in \{1 \ldots |\sigma| - 1\}$ do
        $v := \sigma[i-1]$; $v' := \sigma[i]$;
        check that $\mathrm{transpose}_F(\mathrm{row}[v], \mathrm{row}[v'])$ is a symmetry of $F$;
    return matrix $M$ constructed from row
```

(Description of Algorithm 1). For an illustration, see Figure 4a. The algorithm applies IR for each $v \in \sigma$ (see Figure 4b). All vertices v' in other orbits which are individualized in this process, i.e., which are fixed once v is fixed, are added to the purported "row" of v. We then verify that every row transposition of the resulting matrix is indeed a symmetry of F.

(Correctness of Algorithm 1). We first make the following observation for orbits of stabilizers in row interchangeability groups.

▶ **Lemma 4.** *Let $\Gamma = \mathrm{Sym}(n)$ be a row interchangeability group acting on $[n] \times [m]$. For every $(i, j) \in [n] \times [m]$, the orbit of $(k, l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$ of (i, j) is given by*

$$(k, l)^{\Gamma_{(i,j)}} = \begin{cases} \{(i, l)\} & \text{if } k = i \\ ([n] \setminus \{i\}) \times \{l\} & \text{otherwise.} \end{cases}$$

Proof. Interpreting $[n] \times [m]$ as $n \times m$-matrix M, recall that Γ acts by permuting the rows of M. In other words, the stabilizer $\Gamma_{(i,j)}$ consisting of all row permutations that fix the i-th row and permute the other rows arbitrarily. Now consider the orbit of $(k, l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$. If $k = i$, then (k, l) can only be mapped to elements in the same row as $\Gamma_{(i,j)}$ fixes the i-th row of M. On the other hand, since Γ acts by permuting the rows, every element of M can only be mapped to elements in the same column, that is, (k, l) must be fixed. Similarly, for $k \neq i$, the element (k, l) can be mapped to all elements in the l-th column except for (i, l). ◀

Next, we prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, the algorithm is guaranteed to detect row interchangeability groups.

▶ **Theorem 5.** *Let F be a SAT formula.*

1. *If Algorithm 1 returns a matrix M, every row permutation of M is a symmetry of F.*
2. *If F exhibits row interchangeability with at least three rows including the input set σ and $G(F)$ is a Tinhofer graph, Algorithm 1 detects this structure and returns a corresponding matrix of literals.*

Proof. The first claim is guaranteed by the last part of Algorithm 1 which ensures that transpositions of the rows of the returned matrix M are indeed symmetries of F (Line 11). This implies that arbitrary row permutations are symmetries of F.

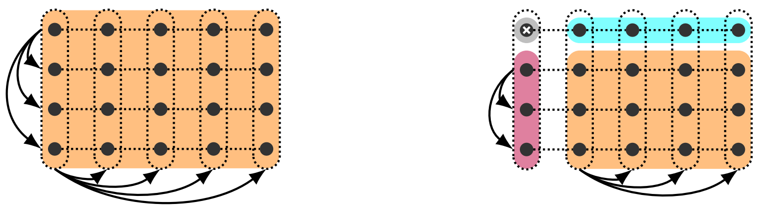
Now assume that F exhibits a row symmetry with at least three rows including σ and $G(F)$ is Tinhofer. We argue that the algorithm successfully detects this symmetry. We remark that the orbits of $\mathrm{Aut}(G(F))$ restricted to the literals are precisely orbits of $\mathrm{Aut}(F)$. Let L be the disjoint direct factor of F containing σ and assume that the literals in L can be partitioned into a matrix M that exhibits row symmetry (see Figure 4a). Due to the assumption that $G(F)$ is Tinhofer, if the vertex v corresponding to a literal l of F is individualized, the resulting refined coloring consists of the orbits of $\mathrm{Aut}(G(F))_v$. In particular, due to Lemma 4, the vertices in the row of M are fixed and all other vertices are contained in orbits of size at least two since we have at least three rows (see Figure 4b). Note that since we have at least three rows, $\neg l$ must be in the row of l. Hence after executing the loop for v, $\mathrm{row}[v]$ contains precisely the vertices in the row of v. Isomorphism-invariance of the IR routine (see Lemma 1) ensures that for each row, the order in which symmetrical singletons are colored will be consistent in each row (see Line 7). This ensures that the rows we construct can indeed be transposed (see Line 11 onwards), and the algorithm correctly returns a corresponding matrix. ◀

Recursive Row Symmetry. In practice, orbits often do not *just* exhibit a row symmetry. In particular, we consider the case that an orbit of size k, with a natural symmetric action, is connected to another orbit of size ck, where the symmetric action acts on blocks of size c (see Figure 4c). We extend our algorithm to detect this particular case as follows: in Line 6, we add fragments of other colors instead of vertices in singletons to the row. Let c be a color of π with a fragment c' in π'. We add the vertices $\pi'^{-1}(c')$ to the row, whenever $|\pi'^{-1}(c')||\sigma| = |\pi^{-1}(c)|$. This means we consider vertices of c', whenever there is the possibility that the color c is split into $|\sigma|$ parts of size $|\pi'^{-1}(c')|$. We call $\pi'^{-1}(c')$ a *block* of its orbit. On these blocks, we call our algorithm for row symmetry recursively. Essentially, this enables us to detect recursive structures of row symmetry.

Row Symmetry in Stabilizer. A slight extension is that if the test for row symmetry fails, we recurse on the largest fragment from the first IR call and check whether it exhibits row symmetry. This extension is used for the other detection algorithms as well.

3.2 Row-Column Symmetry

Next, we describe a detection algorithm for row-column symmetry. As discussed in Section 2.3, a disjoint direct factor exhibiting row-column symmetry consists of an orbit of literals and its negation, which is also an orbit of literals. We detect row-column symmetry only on one of these orbits, and expand the resulting automorphisms to the other one: For a permutation φ of $\mathrm{Lit}(F)$ and all $l \in \mathrm{Lit}(F)$, let


(a) A row-column symmetry with 4 rows and 5 columns.

(b) Individualizing a vertex identifies its row and column.

Figure 5 Illustrations of different aspects of row-column symmetry.

$$\mathrm{expand}_F(\varphi)(l) := \begin{cases} \varphi(l) & \text{if } l \in \mathrm{supp}(\varphi) \\ \neg\varphi(\neg l) & \text{if } \neg l \in \mathrm{supp}(\varphi) \\ l & \text{otherwise.} \end{cases}$$

(Description of Algorithm 2). For an illustration, see Figure 5a. Given a set $\sigma \subseteq \mathrm{Lit}(F)$, we apply IR to a fixed vertex $v \in \sigma$ (see Figure 5b). Assuming that a row-column symmetry is present, this determines a purported "row" $\mathrm{row}[v]$ and "column" $\mathrm{col}[v]$ of v. The algorithm now successively individualizes the vertices in $\mathrm{row}[v]$ and $\mathrm{col}[v]$. This way, every vertex in σ is assigned a reference vertex in each of $\mathrm{row}[v]$ and $\mathrm{col}[v]$, determining its position in the purported matrix. We then verify that the matrix is well-defined and that every row and column transposition, expanded to $\neg\sigma$, is indeed a symmetry of F.

(Correctness of Algorithm 2). In order to prove the correctness of Algorithm 2, we first observe the following:

▶ **Lemma 6.** *Let $\Gamma = \mathrm{Sym}(n) \times \mathrm{Sym}(m)$ be a row-column symmetry group acting on $[n] \times [m]$. For every $(i, j) \in [n] \times [m]$, the orbit of $(k, l) \in [n] \times [m]$ under the action of $\Gamma_{(i,j)}$ is given by*

$$(k,l)^{\Gamma_{(i,j)}} = \begin{cases} \{(k,l)\} & \textit{if } (k,l) = (i,j) \\ \{i\} \times ([m] \setminus \{j\}) & \textit{if } k = i,\ l \neq j \\ ([n] \setminus \{i\}) \times \{j\} & \textit{if } k \neq i,\ l = j \\ ([n] \setminus \{i\}) \times ([m] \setminus \{j\}) & \textit{otherwise.} \end{cases}$$

Proof. We identify $[n] \times [m]$ with the entries of an $n \times m$-matrix M. Then Γ acts on M by permuting the rows and the columns of M. Let $\pi \in \Gamma$ be a permutation that fixes the entry (i, j). Write $\pi = (\pi_r, \pi_c)$, where π_r is a permutation of the rows and π_c a permutation of the columns of M. Then π_r fixes the i-th row and π_c fixes the j-th column of M. On the other hand, every such element of Γ fixes the entry (i, j).

Now consider the orbit of $(k, l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$. By definition, it consists of (k, l) for $(k, l) = (i, j)$. For $k = i$ and $l \neq j$, we can map $(k, l) = (i, l)$ to all elements in the i-row, except for (i, j). Similarly, we argue if $k \neq i$ and $l = j$. Finally, if $k \neq i$ and $l \neq j$, we can map (k, l) to all vertices (k', l') with $k' \neq i$ and $l' \neq j$. This shows the claim. ◀

We prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, it is guaranteed to detect row-column symmetry groups.

Algorithm 2 Detection algorithm for row-column symmetry.

```
function DetectRowColumnSymmetry
    Input : ➢ formula $F$
            ➢ set $\sigma \subseteq \mathrm{Lit}(F)$
    Output : ≺ candidate matrix $M$, or $\bot$ if check fails
    $(G, \pi) := G(F)$, $\pi' := \mathrm{IR}((G, \pi), \varepsilon)$;
    choose arbitrary $v \in \sigma$;
    $\pi_v := \mathrm{IR}((G, \pi), v)$;
    check that $\sigma$ has 4 fragments in $\pi_v$;
    label fragments of $\sigma$ in $\pi_v$ not containing $v$ as $\sigma_1$, $\sigma_2$, $\sigma_3$ in increasing size;

    // we determine "coordinates" in matrix relative to $v$
    $\mathrm{row}[v] := \mathrm{col}[v] = v$ ;                // $v$ defines a row and a column
    foreach $r \in \sigma_1$ do
        $\mathrm{row}[r] := v$, $\mathrm{col}[r] := r$ ;     // $r$ is in row of $v$, and defines a column
        $\pi_r := \mathrm{IR}((G, \pi'), r)$;
        let $\tau$ be the fragment of $\sigma$ in $\pi_r$ of size $|\sigma_2|$ not containing $v$ if exists;
        foreach $t \in \tau$ do $\mathrm{col}[t] := r$;          // $t$ is in column of $r$
    foreach $c \in \sigma_2$ do
        $\mathrm{col}[c] := v$, $\mathrm{row}[c] = c$ ;      // $c$ is in column of $v$, and defines a row
        $\pi_c := \mathrm{IR}((G, \pi'), c)$;
        let $\tau$ be the fragment of $\sigma$ in $\pi_c$ of size $|\sigma_1|$ not containing $v$ if exists;
        foreach $t \in \tau$ do $\mathrm{row}[t] := c$;             // $t$ is in row of $c$
    construct matrix $M$ where $M[r, c] = v'$ with $\mathrm{row}[v'] = r$ and $\mathrm{col}[v'] = c$;

    // verify that $M$ exhibits row-column symmetry
    check that every vertex in $\sigma$ has a unique row and a unique column label;
    check that distinct vertices are assigned distinct label pairs;
    check that $M$ has pairwise disjoint rows, and pairwise disjoint columns;
    foreach $r \in \sigma_1$ do  check that $\mathrm{expand}_F(\mathrm{transpose}_F(M[*, r], M[*, v]))$ is a
     symmetry of $F$;                          // $M[*, x]$ denotes column of $x$
    foreach $c \in \sigma_2$ do  check that $\mathrm{expand}_F(\mathrm{transpose}_F(M[c, *], M[v, *]))$ is a
     symmetry of $F$ ;                            // $M[x, *]$ denotes row of $x$
    return $M$
```

▶ **Theorem 7.** *Let F be a SAT formula.*

1. *If Algorithm 2 returns a matrix M of literals, every permutation of the rows or the columns of M, expanded to the negations of the literals in M, is a symmetry of F.*
2. *If F exhibits a row-column symmetry with at least three rows and at least three columns including σ and $G(F)$ is a Tinhofer graph, then Algorithm 2 detects this structure and returns a corresponding matrix representation of the literals in σ.*

Proof. The first claim is guaranteed by the last part of Algorithm 2 which ensures that transpositions of the rows (Line 24) and columns (Line 25) of the returned matrix M, expanded to the corresponding negated literals, are indeed symmetries of F. By suitably composing such transpositions, we obtain that every permutation of the rows or columns of M induces a symmetry of F in this way.

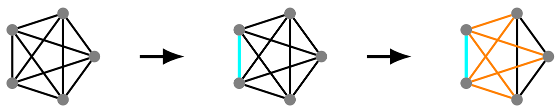


Figure 6 Individualizing a variable v in a Johnson symmetry (represented by edges in the illustration), splits the set of edges into edges incident to v, and edges not incident to v.

Now assume that $G(F)$ is Tinhofer and that F exhibits row-column symmetry with at least three rows and columns on σ. In other words, the literals in σ can be arranged in a matrix M on which $\mathrm{Aut}(F)$ acts by row and column permutations (see Figure 5a). Individualizing a fixed vertex $v \in \sigma$ causes σ to split into four fragments according to the orbits of the stabilizer $\mathrm{Aut}(F)_v$: the singleton $\{v\}$, two fragments σ_1 and σ_2 corresponding to the remainders of the row and the column of M containing v, and a fragment σ_3 containing the remaining vertices (see Lemma 6 Figure 5b). Since we assume that M has at least three rows and columns, $\sigma_1, \sigma_2, \sigma_3$ are non-singletons and σ_3 is the largest fragment. Without loss of generality, let $\sigma_1 \cup \{v\}$ be the row and $\sigma_2 \cup \{v\}$ be the column of v in M. Every column of M is determined by the unique element of $\sigma_1 \cup \{v\}$ that it contains (similarly for the rows). Individualizing a vertex $r \in \sigma_1$ leads a similar split of σ into four fragments. The fragments corresponding to the row and column of r can be distinguished by observing that v lies in the same row, but not in the same column as r. For all vertices in the column of r, we store this information (Line 13). Similarly, we proceed for the columns (Line 18). After this procedure, every element of σ is assigned a row and column representative in σ_2 and σ_1 respectively, which, up to a permutation of the rows and columns, allows us to recover the matrix M. ◀

3.3 Johnson Symmetry

Finally, we describe a procedure to detect Johnson actions. We remark that there is a classic algorithm to detect Johnson groups [10]. A difference to our heuristic is that we do not know the generators of the group, and instead apply techniques directly on a given graph.

Our aim is to identify the variables in the input set σ with the 2-subsets of $[n]$, where $|\sigma| = \binom{n}{2}$. We thus search for a bijection $b\colon \sigma \to \binom{[n]}{2}$ such that $\mathrm{Aut}(F)$ acts as the Johnson group $\mathcal{J}_n$ on σ via this bijection (see Section 2.3, Figure 2, and Figure 3). To avoid confusion, we refer to the elements of $[n]$ as *labels* and to those of $\mathrm{Lit}(F)$ as *literals* or *vertices* of $G(F)$.

(Description of Algorithm 3.) Suppose that F exhibits a Johnson symmetry on σ. As described above, there is an (unknown) bijection $b\colon \sigma \to \binom{[n]}{2}$ (see Figure 3). We maintain a list $\mathrm{label}[v]$ for every $v \in \sigma$, to which we add $i \in [n]$ when we deduce that $i \in b(v)$. If the algorithm returns a list label, a possible bijection b is given by $b(v) = \mathrm{label}[v]$ for all $v \in \sigma$. Note that b is only determined up to permutation of the labels, so our algorithm merely determines vertices obtaining the same label and assigns the labels consecutively.

The algorithm proceeds as follows: we apply IR to $v \in \sigma$, yielding a coloring π_v. Write $b(v) = \{i,j\}$ for some $i,j \in [n]$. The coloring π_v has three fragments: $\{v\}$, the fragment σ_v containing all $u \in \sigma$ with $|b(u) \cap \{i,j\}| = 1$, and the remaining elements (see Figure 6). We call the vertices in σ_v *adjacent* to v and collect them in $\mathrm{ad}[v]$. Now choose $w \in \mathrm{ad}[v]$. We can assume $b(w) = \{j,k\}$ for some $k \notin \{i,j\}$. As before, we find the vertices adjacent to w by applying IR to w. Individualizing both v and w, the resulting coloring $\pi_{v,w}$ contains exactly one further singleton consisting of $y \in \sigma$ with $b(y) = \{i,k\}$. Now $\mathrm{ad}[v] \cap \mathrm{ad}[w] = \{y\} \cup \{u \in \sigma\colon b(u) = \{j,r\} \text{ for some } r \notin \{i,j,k\}\}$. The vertices in $\mathrm{ad}[v] \cap \mathrm{ad}[w] \setminus \{y\}$ thus obtain the label j. Similarly, we determine the vertices obtaining the label i or k. After ensuring that the labels have not been considered previously, we add them to the list label for the respective vertices.

Algorithm 3 Detection algorithm for Johnson actions.

```
1  function DetectJohnson
       Input : ➢ formula $F$
               ➢ set $\sigma \subseteq \mathrm{Lit}(F)$
       Output : ≺ bijective labeling of $\sigma$ by 2-subsets of $[n]$, or $\perp$ if check fails
2      $(G, \pi) \coloneqq G(F)$, $\pi' \coloneqq \mathrm{IR}((G, \pi), \varepsilon)$;
3      check that $|\sigma| \geq 28$ and $|\sigma| = \binom{n}{2}$ for some $n \in \mathbb{N}$;
4      foreach vertex $v \in \sigma$ do set label$[v] = []$;
5      vnr $= 1$;
6      while there are vertices $v \in \sigma$ with $|label[v]| \leq 1$ do
7          $E_i \coloneqq E_j \coloneqq E_k \coloneqq \{\}$;
8          choose $v \in \sigma$ with $|\mathrm{label}[v]| \leq 1$;
9          $\pi_v \coloneqq \mathrm{IR}((G, \pi'), v)$;
10         check that number of fragments of $\sigma$ in $\pi_v$ is 3;
11         let $\sigma_v$ be the smaller non-singleton fragment;
12         foreach $x \in \sigma_v$ do add $x$ to ad$[v]$ ;
13         choose arbitrary $w \in$ ad$[v]$;
14         $\pi_w \coloneqq \mathrm{IR}((G, \pi'), w)$;
15         check that number of fragments of $\sigma$ in $\pi_w$ is 3;
16         let $\sigma_w$ be the smaller non-singleton fragment;
17         foreach $x \in \sigma_w$ do add $x$ to ad$[w]$;
18         $\pi_{v,w} \coloneqq \mathrm{IR}((G, \pi_v), w)$;
19         let $\{y\}$ be the unique singleton fragment of $\sigma$ in $\pi_{v,w}$ different from $\{v\}$ and
            $\{w\}$ if existent, otherwise return $\perp$;
20         $\pi_y \coloneqq \mathrm{IR}((G, \pi'), y)$;
21         check that number of fragments of $\sigma$ in $\pi_y$ is 3;
22         let $\sigma_y$ be the smaller non-singleton fragment;
23         foreach $x \in \sigma_y$ do add $x$ to ad$[y]$;
24         add $v$ to $E_i$ and $E_j$, add $w$ to $E_j$ and $E_k$, add $y$ to $E_i$ and $E_k$;
25         foreach $x \in ad[v] \cap ad[y]$ and $x \neq w$ do add $x$ to $E_i$;
26         foreach $x \in ad[v] \cap ad[w]$ and $x \neq y$ do add $x$ to $E_j$;
27         foreach $x \in ad[w] \cap ad[y]$ and $x \neq v$ do add $x$ to $E_k$;
28         foreach $E \in \{E_i, E_j, E_k\}$ do
29             if $\bigcap_{v \in E} label[v] = \emptyset$ then
30                 append vnr to label$[v]$ for $v \in E$;
31                 vnr $+= 1$;
32         check that new labels were added to label in this iteration;
33     // verify that $F$ exhibits Johnson symmetry
34     verify that label induces a bijection between $\sigma$ and $\binom{[n]}{2}$;
35     foreach $i \in [n-1]$ do
36         let $\beta$ denote the permutation of $\sigma$ induced by the Johnson action induced by
            $(i, i+1) \in \mathrm{Sym}(n)$ using label;
37         check that $\mathrm{expand}_F(\beta)$ is a symmetry of $F$;
38     return label
```

(Description of Algorithm 3). We again make some observations about stabilizers in Johnson groups:

▶ **Lemma 8.** *Let $n \in \mathbb{N}$ and consider the Johnson group $\Gamma := \mathcal{J}_n$, acting on 2-subsets of $[n]$.*

1. *For $\{i,j\} \in \binom{[n]}{2}$, the orbit of $S \in \binom{[n]}{2}$ under the stabilizer $\Gamma_{\{i,j\}}$ of $\{i,j\} \in \binom{[n]}{2}$ is given by*

$$S^{\Gamma_{\{i,j\}}} = \begin{cases} \{S\} & \text{if } S = \{i,j\} \\ \{T \in \binom{[n]}{2} \colon |T \cap \{i,j\}| = 1\} & \text{if } |S \cap \{i,j\}| = 1 \\ \{T \in \binom{[n]}{2} \colon T \cap \{i,j\} = \emptyset\} & \text{if } S \cap \{i,j\} = \emptyset. \end{cases}$$

2. *For $k \neq i,j$, the orbit of $S \in \binom{[n]}{2}$ under $\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}$ is given by*

$$S^{\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}} = \begin{cases} \{S\} & \text{if } S \in \{\{i,j\},\{i,k\},\{j,k\}\} \\ \{\{i,r\} \colon r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{i,s\} \text{ for some } s \in [n] \setminus \{i,j,k\} \\ \{\{j,r\} \colon r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{j,s\} \text{ for some } s \in [n] \setminus \{i,j,k\} \\ \{\{k,r\} \colon r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{k,s\} \text{ for some } s \in [n] \setminus \{i,j,k\} \\ \{S \in \binom{[n]}{2} \colon S \cap \{i,j,k\} = \emptyset\} & \text{if } S \cap \{i,j,k\} = \emptyset. \end{cases}$$

Proof.

1. Let $S \in \binom{[n]}{2}$. If $S = \{i,j\}$, the orbit $S^{\Gamma_{\{i,j\}}}$ consists only of S by definition of the stabilizer. Now suppose that $|S \cap \{i,j\}| = 1$ holds. Without loss of generality, let $S = \{i,r\}$ for some $r \in [n] \setminus \{i,j\}$. Let $\pi \in \Gamma_{\{i,j\}}$. Either π fixes i and j, in which case we have $S^\pi = \{i,r'\}$ for some $r' \in [n] \setminus \{i,j\}$, or π interchanges i and j, in which case we have $S^\pi = \{j,r'\}$ for some $r' \in [n] \setminus \{i,j\}$. In both cases, we have $|S^\pi \cap \{i,j\}| = 1$. On the other hand, it is easy to see that for every set $T \in \binom{[n]}{2}$ with $|T \cap \{i,j\}| = 1$, there exists $\pi \in \Gamma_{\{i,j\}}$ with $S^\pi = T$. The description of $S^{\Gamma_{\{i,j\}}}$ in the case $S \cap \{i,j\} = \emptyset$ can be derived analogously.
2. Note that an element in $\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}$ fixes or interchanges the labels i and j, and at the same time fixes or interchanges the labels i and k. This is only possible if it fixes all of i, j and k. The structure of the orbits then follows similarly to the first claim. ◀

We now prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, the algorithm is guaranteed to detect that F exhibits a Johnson symmetry on the input set σ.

▶ **Theorem 9.** *Let F be a SAT formula.*

1. *If Algorithm 3 returns a list* label *of labels in $[n]$, then for every element in $\mathcal{J}_n$, the induced permutation of σ according to label, expanded to $\neg\sigma$, is a symmetry of F.*
2. *If F exhibits a Johnson symmetry with Johnson group $\mathcal{J}_n$ with $n \geq 8$ on σ and $G(F)$ is a Tinhofer graph, then Algorithm 3 detects this structure and returns a corresponding labeling of the literals in σ by 2-subsets of $[n]$.*

Proof. The last part of Algorithm 3 ensures that the Johnson action j_π induced by a transposition $\pi := (i, i+1) \in \mathcal{J}_n$ by permuting the elements in σ according to their labels in *label* is a symmetry of F when expanded to $\neg\sigma$. By suitably composing these transpositions, it follows that every element of $\mathcal{J}_n$ induces a symmetry of F in this way.

Now suppose that F exhibits a Johnson symmetry with Johnson group $\mathcal{J}_n$ with $n \geq 8$ (i.e., $|\sigma| \geq 28$). Furthermore, assume that $G(F)$ is Tinhofer. In particular, there is a bijection $b \colon \sigma \to \binom{[n]}{2}$ (see Figure 3). We claim that when the algorithm terminates, there is a permutation $\tau \in \mathrm{Sym}(n)$ of the label set $[n]$ such that we have $\mathrm{label}[v] = \{\tau(i), \tau(j)\}$ if $b(v) = \{i, j\}$. Note that the bijection b itself is determined only up to permutation of the labels. Again, for the sake of clarity, we refer to the elements of $[n]$ as *labels* and reserve the term *vertices* for the vertices of the graph $G(F)$.

The individualization of a vertex v with $b(v) = \{i, j\}$ (Line 9) leads to a color partition with three fragments since $G(F)$ is Tinhofer (see Lemma 8 and Figure 6). The smaller non-singleton fragment is $\sigma_v = \{u \in \sigma \colon |b(u) \cap \{i, j\}| = 1\}$. For this, note that $|\sigma_v| = 2(n-2)$ holds and that we have $n \geq 8$ by assumption. The list $\mathrm{ad}[v]$ (Line 12) then consists of all vertices $u \in \sigma$ with $b(u) = \{i, r\}$ or $b(u) = \{j, r\}$ with $r \in [n] \setminus \{i, j\}$.

Now let $w \in \mathrm{ad}[v]$. Up to this point, the labels i and j are interchangeable, so we may assume $b(w) = \{j, k\}$ for some $k \in [n] \setminus \{i, j\}$. We repeat the above procedure with w in place of v. In particular, $\mathrm{ad}[w]$ (Line 17) contains all vertices $u \in \sigma$ with $b(u) = \{j, r\}$ or $b(u) = \{k, r\}$ for $r \in [n] \setminus \{j, k\}$.

Finally we individualize both v and w to obtain the coloring $\pi_{v,w}$. The fragments are given by Lemma 8. In particular, we obtain $b(y) = \{i, k\}$. Apart from y, the intersection $\mathrm{ad}[v] \cap \mathrm{ad}[w]$ contains all vertices $u \in \sigma$ with $b(u) = \{j, r\}$ for $r \in [n] \setminus \{i, j, k\}$, and we add them to E_j (Line 26). Similarly, we construct the sets E_i and E_k (Lines 25 and 27).

From this explicit description, it is clear that $u \in \sigma$ is added to E_i precisely if $i \in b(u)$ (similarly for E_j and E_k). In particular, for distinct vertices $u_1, u_2 \in E_i$, we have $b(u) \cap b(v) = \{i\}$. Thus if the lists $\mathrm{label}[u]$ for $u \in E_i$ have a common entry, the label i has been considered before (recall that $|E_i| > 1$ holds). Otherwise, we add the current vertex number *vnr* to $\mathrm{label}[u]$ for all $u \in E_i$ (Line 28) and set $\tau(i) = \mathrm{vnr}$. This way, $\mathrm{label}[u]$ remains duplicate-free and only ever contains labels $\tau(l)$ for $l \in b(u)$. In particular, we always maintain the property $|\mathrm{label}[u]| \leq 2$. In each iteration of the while loop, one of the labels i and j was not considered before (due to $|\mathrm{label}[v]| \leq 1$). In particular, the loop is executed at most n times. When it stops, we have $|\mathrm{label}[v]| = 2$ for all vertices v. ◀

Johnson Action on Row Symmetry. Quite commonly, SAT instances which search for a graph, will search for a graph with a certain *vertex property*. For example, when asking for a k-colorable graph, there will be (interchangeable) colors attached to each vertex of the graph. In order to detect a corresponding symmetry structure, we want to detect blocks which correspond to the labels in the Johnson domain. The detection works by stabilizing vertices in other orbits, and checking whether they split apart the Johnson orbit precisely into the vertices marked with a particular label, and a remainder. If so, these blocks are collected and considered in our overall Johnson action. Finally, we run row symmetry detection on the collected blocks.

4 Implementation

We now give an overview of our new symmetry breaking tool SATSUMA. The input of our algorithm is a CNF formula F. The output is a symmetry breaking constraint for F. We first discuss the breaking constraints produced for a given detected structure.

Breaking Constraints. We produce lex-leader constraints for each detected structure: we use precisely the automorphisms constructed in Algorithm 1, Algorithm 2, and Algorithm 3. Before we can produce lex-leader constraints, we must however fix an ordering on the variables. The ordering used for matrix models simply orders the matrix row-by-row. For Johnson groups, we begin with the vertices of the first label (see Algorithm 3), then the remaining vertices of the second label, and so forth.

family		CMS		BreakID+CMS			satsuma+CMS		
name	size	solved	avg	prep	solved	avg	prep	solved	avg
channel	10	2	484.99	4.727	**10**	**0.032**	**0.404**	**10**	0.033
cliquecolor	20	2	574.734	0.129	13	228.998	**0.058**	**20**	**0.845**
coloring	55	21	377.338	42.12*	26	317.32	**1.071**	**27**	**307.632**
fpga	10	6	321.596	0.035	**10**	0.01	**0.01**	**10**	**0.008**
md5	11	5	358.616	0.635	5	359.382	**0.548**	**6**	**349.171**
php	10	3	423.266	6.337	**10**	0.043	**0.128**	**10**	**0.036**
ramsey	7	2	428.613	1.681	3	343.086	**0.394**	**5**	**235.27**
urquhart	6	**6**	0.768	0.14	**6**	**0.008**	**0.032**	**6**	0.066

Figure 7 Benchmarks comparing BreakID to satsuma, using the solver CryptoMiniSAT (CMS). The timeout is 600 seconds, all times are given in seconds. The columns "prep" denote the average time used for symmetry breaking. Columns "solved" refer to the number of solved instances by CMS, and "avg" is the average time spent by CMS. *BreakID could not compute the symmetry breaking constraints of two coloring instances within the timeout. We declared these as a timeout for the SAT benchmarks (but the other configurations also timed out on these instances).

High-level Algorithm. The high-level algorithm proceeds as follows:

(Step 1). Construct a model graph from the given CNF formula.

(Step 2). Run the algorithms described in the previous section in the following order: Johnson groups (Algorithm 3), row-column symmetry (Algorithm 2), row interchangeability (Algorithm 1). Whenever a structure is found, all orbits covered by the structure are marked. The subsequent analysis only considers *unmarked* orbits. For each structure, symmetry breaking constraints are constructed as described above. Lastly, we maintain a vertex coloring of the model graph, which we call the *remainder coloring*: this coloring restricts the symmetries of the model graph to symmetries not yet covered by detected structures.

(Step 3). Run symmetry detection on the graph colored with the remainder coloring. Then, the *binary clause* heuristic of BreakID is applied for all variables not yet ordered by already produced lex-leader constraints: a stabilizer chain of the automorphism group is approximated, and for each stabilized variable x a short lex-leader constraint for each other literal y of its orbit is produced, i.e., essentially the binary constraint $x \leq y$ (see [21] for a detailed description). Lastly, a lex-leader constraint for each generator is produced.

Implementation. The tool is written in C++, and is freely available as open source software [2]. The tool dejavu [3, 4, 6] is used for providing general-purpose symmetry detection, the individualization-refinement framework, and data structures for symmetries. Significant parts of the implementation, in particular the generation of lex-leader constraints and binary clauses, are reverse-engineered from BreakID. Our reimplementation of these routines differs in two crucial aspects from the original one: first, BreakID uses the symmetry detection tool saucy [18] instead of dejavu. Second, we use different data structures and algorithms for the handling of symmetry.

5 Benchmarks

We compare the state-of-the-art static symmetry breaking tool BreakID (version 2.6) to satsuma.

As SAT solvers, we use CryptoMiniSAT [39] and CaDiCaL [12]. The benchmarks using CaDiCaL largely concur with the CryptoMiniSAT benchmarks, and our descriptions will focus on the results using CryptoMiniSAT. The timeout for all benchmarks is 600

family		CaD		BreakID+CaD			satsuma+CaD		
name	size	solved	avg	prep	solved	avg	prep	solved	avg
channel	10	2	494.226	4.727	**10**	**0.077**	**0.404**	**10**	**0.077**
cliquecolor	20	9	442.373	0.129	13	216.999	**0.058**	**20**	**0.2**
coloring	55	20	393.864	42.12*	26	316.779	**1.071**	**28**	**301.783**
fpga	10	5	391.29	0.035	**10**	**0.008**	**0.01**	**10**	0.025
md5	11	**6**	339.378	0.635	**6**	343.324	**0.548**	**6**	**324.716**
php	10	3	422.976	6.337	**10**	**0.085**	**0.128**	**10**	0.1
ramsey	7	2	428.583	1.681	3	342.908	**0.394**	**5**	**192.299**
urquhart	6	2	449.622	0.14	**6**	**0.005**	**0.032**	**6**	0.052

Figure 8 Benchmarks comparing BreakID to satsuma. The SAT solver used is CaDiCaL (CaD). The timeout used is 600 seconds. The columns "prep" refer to the time in seconds used to compute the symmetry breaking constraint. Columns "solved" refer to the number of solved instances by CaD, and "avg" is the average time used by CaD (*excluding* the time used for symmetry breaking). *BreakID could not compute the symmetry breaking constraints of two coloring instances within the timeout.

seconds. We separately measure the time spent on symmetry breaking itself, and SAT solving. All benchmarks ran sequentially on an Intel Core i7 9700K with 64GB of RAM on Ubuntu 20.04.

Benchmark Instances. We run benchmarks on a variety of well-established instance families exhibiting symmetry (see Figure 7). The sets coloring, urquhart, fpga, md5, and channel are part of the distribution of BreakID [21]. We generate pigeonhole principle (php) instances, Ramsey instances, and clique coloring instances using the tool cnfgen [31]. The set of parameters for clique coloring is similar to [26], but we added larger instances. All instances are unsatisfiable.

Regarding the detected symmetry structures of these instances, we detect Johnson symmetry on the ramsey and cliquecolor families. On php, channel, and fpga, satsuma detects row-column symmetry, and BreakID corresponding row interchangeability (see also [36, 21]). The coloring instances exhibit a variety of different symmetries, but in particular also row symmetry [21]. In urquhart and md5, no structure is detected by either of the tools.

Regarding our choice of benchmark instances, we stress that our main goal is to observe whether detecting richer structures can improve performance compared to existing approaches.

SAT Benchmarks. An overview of the results can be found in Figure 7 (for CaDiCaL, see Figure 8). Considering the results, we observe that satsuma solves more instances, and solving times are considerably lower on average on the cliquecolor and ramsey instances. We recall that these instance families exhibit Johnson symmetry. On all sets with row and row-column symmetry, that is channel, coloring, fpga, and php, we observe that solved instances and average solving times are comparable. On coloring, we observe that satsuma solves one more instance than BreakID (and two more using CaDiCaL). For urquhart, both satsuma and BreakID rely on the binary clause strategy. The results indicate that BreakID is more effective in breaking symmetry, which is however outweighed by the faster runtime of satsuma. The md5 instances only contain a single non-trivial symmetry. Here, satsuma produces more breaking clauses, and we observe a consistent albeit marginal speedup. It should be mentioned that it does however seem plausible that the observed speed-up may be due to shuffling of literals in clauses, or other factors.

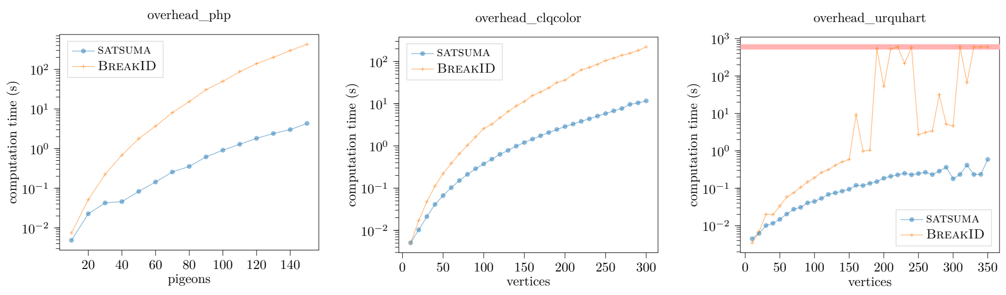


Figure 9 Benchmarks comparing the computational overhead of BreakID to satsuma. The shown computation time is the time spent computing symmetry breaking constraints for an instance using the respective tool. The red bar indicates the timeout of 600 seconds.

In particular, we point out that satsuma compares favorably on instance families which exhibit Johnson symmetry. We believe this to be due to our detection of Johnson symmetry and the subsequent generation of more favorable constraints. Crucially, on all successfully solved instances of cliquecolor and ramsey, the *remainder contains no symmetry*: all symmetries are detected and in turn broken solely using the algorithms of this paper, and no general-purpose symmetry detection and breaking is applied.

We observe that the average time spent computing the symmetry breaking constraints is lower on all families for satsuma. A more in-depth analysis follows below.

Computational Overhead. We conduct further benchmarks to gauge the computational overhead incurred by BreakID and satsuma. We test three different benchmark families: php, cliquecolor, and urquhart (generated using cnfgen). For php, we increase the number of pigeons from 10 to 150 (with $n - 1$ holes, respectively). For cliquecolor, we increase the number of vertices of the prospective graph from 10 to 300 (the size of the clique is 3 and number of colors 2). In urquhart, we use random 5-regular graphs, increasing the number of vertices from 10 to 350. We chose these instance families such that they cover the different symmetry detection routines in satsuma: the family php essentially measures the runtime of our row-column routine, cliquecolor that of the Johnson routine, and urquhart uses general purpose symmetry detection, followed by the binary clause strategy.

Figure 9 summarizes the results. In all instance families, the data suggest that satsuma asymptotically scales better than BreakID. These results match our observations regarding overhead from the first part of the benchmarks (see Figure 7).

We believe there are multiple reasons why satsuma runs faster than BreakID. First, our new algorithms of Section 3 verify symmetries on the CNF formula instead of the model graph. This is advantageous because symmetries of the CNF only explicitly map literals, whereas symmetries of the model graph also explicitly map clauses. Second, most routines in our implementation run proportional in the size of the *support* of symmetries, as opposed to the number of literals of F. Third, for general-purpose symmetry detection, dejavu seems to be more efficient in computing automorphism groups of SAT instances than saucy [3].

We mention that in the urquhart instances, the outliers with high running time seem to be due to saucy taking a long time to compute symmetries for BreakID. On the other hand, in these cases, we observe that the symmetries as returned by dejavu are less suitable for the binary clause heuristic, leading to fewer produced clauses. This could however be easily alleviated by a strengthening of the heuristic (e.g., by sometimes applying the Schreier-Sims algorithm for stabilizers as already pointed out in [21]).

6 Conclusions and Future Work

We described a new structure-based approach to symmetry breaking, and demonstrated the effectiveness of our implementation SATSUMA. There seem to be many promising directions in which the present work could be expanded:

- Detect more group structures: in particular, a more generic approach to detect aggregates of groups would be of great interest. Another interesting case might be the symmetries of the family urquhart, which are isomorphic to C_2^k and have been studied previously [32].
- Consider other breaking approaches for certain group structures. So far, we used the knowledge of group structures to pick out automorphisms, for which off-the-shelf lex-leader constraints are generated. Since optimal handling of row-column symmetry and Johnson symmetry seems infeasible with lex-leader constraints [32], other breaking constraints could lead to better results. Moreover, Johnson symmetry allows the use of symmetry reduction developed specifically for graph generation [15, 16, 30].
- Improved techniques for handling of the "remainder". As already pointed out in [21], one potential direction would be to apply the random Schreier-Sims algorithm [38] to produce more small symmetry breaking clauses.
- An enticing feature is proof-logging, as was recently introduced to BREAKID [13].
- The new detection algorithms could be applied in other domains as well: for example, seeing as row interchangeability is successfully used in MIP, it seems only natural that MIP instances may also contain richer structures.
- Sometimes symmetries are not present in a compiled CNF of a given problem (as, e.g., analyzed in [26]). A possible remedy is to allow the user to provide an auxiliary graph that models the original symmetry (see [26]), and the methods proposed in this paper should generalize to this setting.

References

1 Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 836–839. ACM, 2003. doi:10.1145/775832.776042.

2 Markus Anders, Sofia Brenner, and Gaurav Rattan. satsuma. Software, version 1.0., swhId: swh:1:dir:134ea7952a25d92ba2addf7ff25aa18183550735 (visited on 2024-08-05). URL: https://github.com/markusa4/satsuma.

3 Markus Anders and Pascal Schweitzer. dejavu. https://automorphisms.org.

4 Markus Anders and Pascal Schweitzer. Parallel computation of combinatorial symmetries. In *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ESA.2021.6.

5 Markus Anders, Pascal Schweitzer, and Mate Soos. Algorithms transcending the SAT-symmetry interface. In *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 1:1–1:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.1.

6 Markus Anders, Pascal Schweitzer, and Julian Stieß. Engineering a preprocessor for symmetry detection. In *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain*, volume 265 of *LIPIcs*, pages 1:1–1:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.1.

7 Vikraman Arvind, Johannes Köbler, Gaurav Rattan, and Oleg Verbitsky. Graph isomorphism, color refinement, and compactness. *Comput. Complex.*, 26(3):627–685, 2017. doi:10.1007/s00037-016-0147-6.

8 Gilles Audemard, Saïd Jabbour, and Lakhdar Sais. Symmetry breaking in quantified boolean formulae. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2262–2267, 2007. URL: `http://ijcai.org/Proceedings/07/Papers/364.pdf`.

9 László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 684–697. ACM, 2016. `doi:10.1145/2897518.2897542`.

10 László Babai, Eugene M. Luks, and Ákos Seress. Permutation groups in NC. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 409–420. ACM, 1987. `doi:10.1145/28395.28439`.

11 Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. `doi:10.1007/s00224-016-9686-0`.

12 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

13 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.*, 77:1539–1589, 2023. `doi:10.1613/jair.1.14296`.

14 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Çağrı Uluç Yıldırımoğlu. BreakID-kissat in SAT competition 2023 (system description). In *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, Department of Computer Science Series of Publications B, Finland, 2023. Department of Computer Science, University of Helsinki.

15 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries in graphs: The nauty way. In *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 157–172. Springer, 2016. `doi:10.1007/978-3-319-44953-1_11`.

16 Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints An Int. J.*, 24(1):1–24, 2019. `doi:10.1007/s10601-018-9294-5`.

17 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*, pages 148–159. Morgan Kaufmann, 1996.

18 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 530–534. ACM, 2004. `doi:10.1145/996566.996712`.

19 Jo Devriendt and Bart Bogaerts. BreakID: Static symmetry breaking for ASP (system description). *CoRR*, abs/1608.08447, 2016. `doi:10.48550/arXiv.1608.08447`.

20 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2017. `doi:10.1007/978-3-319-66263-3_6`.

21 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *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*, pages 104–122. Springer, 2016. `doi:10.1007/978-3-319-40970-2_8`.

22 Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 49–56. IEEE Computer Society, 2012. doi:10.1109/ICTAI.2012.16.

23 Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming – CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002. doi:10.1007/3-540-46135-3_31.

24 Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kızıltan, Ian Miguel, and Toby Walsh. Matrix modelling. Technical Report APES-36-2001, APES group (2001), 2001.

25 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006. doi:10.1016/S1574-6526(06)80014-3.

26 Tommi A. Junttila, Matti Karppa, Petteri Kaski, and Jukka Kohonen. An adaptive prefix-assignment technique for symmetry reduction. *J. Symb. Comput.*, 99:21–49, 2020. doi:10.1016/j.jsc.2019.03.002.

27 Tommi A. Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer, 2011. doi:10.1007/978-3-642-19754-3_16.

28 George Katsirelos, Nina Narodytska, and Toby Walsh. On the complexity and completeness of static constraints for breaking row and column symmetry. In *Principles and Practice of Constraint Programming – CP 2010 – 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2010. doi:10.1007/978-3-642-15396-9_26.

29 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. A SAT attack on rota's basis conjecture. In *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 4:1–4:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.4.

30 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In *27th International Conference on Principles and Practice of Constraint Programming, CP*, volume 210 of *LIPIcs*, pages 34:1–34:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.34.

31 Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017. doi:10.1007/978-3-319-66263-3_30.

32 Eugene M. Luks and Amitabha Roy. The complexity of symmetry-breaking formulas. *Ann. Math. Artif. Intell.*, 41(1):19–45, 2004. doi:10.1023/B:AMAI.0000018578.92398.10.

33 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

34 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in SAT solving. In *Tools and Algorithms for the Construction and Analysis of Systems – 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018. doi:10.1007/978-3-319-89960-2_6.

35 Marc E. Pfetsch and Thomas Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Math. Program. Comput.*, 11(1):37–93, 2019. `doi:10.1007/s12532-018-0140-y`.

36 Ashish Sabharwal. Symchaff: exploiting symmetry in a structure-aware satisfiability solver. *Constraints An Int. J.*, 14(4):478–505, 2009. `doi:10.1007/s10601-008-9060-1`.

37 Karem A. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 509–570. IOS Press, 2021. `doi:10.3233/FAIA200996`.

38 Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003. `doi:10.1017/CBO9780511546549`.

39 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing – SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 – July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. `doi:10.1007/978-3-642-02777-2_24`.

40 Gottfried Tinhofer. A note on compact graphs. *Discret. Appl. Math.*, 30(2-3):253–264, 1991. `doi:10.1016/0166-218X(91)90049-3`.

The Relative Strength of #SAT Proof Systems*

Olaf Beyersdorff
Friedrich Schiller University Jena, Germany

Johannes K. Fichte
Linköping University, Sweden

Markus Hecher
Massachusetts Institute of Technology, Cambridge, MA, USA

Tim Hoffmann
Friedrich Schiller University Jena, Germany

Kaspar Kasche
Friedrich Schiller University Jena, Germany

Abstract

The propositional model counting problem #SAT asks to compute the number of satisfying assignments for a given propositional formula. Recently, three #SAT proof systems kcps (knowledge compilation proof system), MICE (model counting induction by claim extension), and CPOG (certified partitioned-operation graphs) have been introduced with the aim to model #SAT solving and enable proof logging for solvers.

Prior to this paper, the relations between these proof systems have been unclear and very few proof complexity results are known. We completely determine the simulation order of the three systems, establishing that CPOG simulates both MICE and kcps, while MICE and kcps are exponentially incomparable. This implies that CPOG is strictly stronger than the other two systems.

2012 ACM Subject Classification Theory of computation → Proof complexity; Theory of computation → Proof theory; Theory of computation → Logic and verification; Theory of computation → Automated reasoning

Keywords and phrases Model Counting, #SAT, Proof Complexity, Proof Systems, Lower Bounds, Knowledge Compilation

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.5

Funding *Olaf Beyersdorff*: Carl-Zeiss Foundation and DFG grant BE 4209/3-1
Johannes K. Fichte: ELLIIT (Excellence Center at Linköping – Lund in Information Technology) funded by the Swedish government
Markus Hecher: Austrian Science Fund (FWF) grant J4656, Vienna Science and Technology Fund (WWTF) grant ICT19-065, and Society for Research Funding Lower Austria (GFF) grant ExzF-0004
Tim Hoffmann: Carl-Zeiss Foundation
Kaspar Kasche: Carl-Zeiss Foundation

Acknowledgements Parts of the work has been carried out while the first four authors were visiting the Simons Institute for Theory of Computation at UC Berkeley.

1 Introduction

The *propositional model counting problem* #SAT asks to compute the number of satisfying assignments for a given propositional formula [1]. The #SAT framework allows to efficiently encode and solve many real-world problems from areas such as probabilistic reasoning [5, 45],

* Authors are given in alphabetical order. Parts of the work has been carried out while the first four authors were visiting the Simons Institute for Theory of Computation at UC Berkeley.

risk analysis [26, 59] and explainable artificial intelligence [6, 51]. Interestingly, #SAT is among the hardest combinatorial problems and known to be #P-complete [5, 49, 55]. To put this in relation, by Toda's Theorem [53] any problem from the polynomial hierarchy (PH) can be solved in polynomial time by access to a #SAT oracle. In comparison, the SAT problem is on the first level of PH [20].

Over the last two decades, researchers and solver engineers improved effective *practical #SAT solving* [31] with numerous available #SAT solvers using conceptually quite different approaches. An annual competition captures current trends of solvers and novel practical algorithms, but also reveals that correctness needs to be improved [27].

In contrast to these practical advances, little is known theoretically on the power and limitations of #SAT solving. In both SAT and quantified Boolean formulas (QBF), the main theoretical approach towards gauging the strength of SAT and QBF solvers is through *proof systems and proof complexity* [13, 16]. The relation between proofs and solving is important in at least two aspects.

Firstly, proof systems can *model aspects of solving.* A seminal result in this direction is that CDCL solvers – the predominant approach in SAT solving – tightly correspond to propositional resolution [4, 7, 48], in the sense that any (non-deterministic) CDCL run on an unsatisfiable formula can be efficiently translated into a resolution refutation of the formula and vice versa. Further results are known for practical CDCL [56] and relations between QBF solving and related proof systems [10, 36]. This allows to apply the plethora of proof complexity results e.g. for propositional and QBF resolution [9, 43, 50] to the analysis of solvers. For example any lower bound for proof size in propositional resolution directly translates into a lower bound for CDCL runtime.

Secondly, proof systems can be employed for *proof logging and certifying solver correctness* by designing certified tools. Therefore, formal proof systems are introduced where a practical proof can be efficiently verified by a relatively simple method and easily emitted during solving. Different proof systems and formats have been designed including RUP, RAT and DRAT [29, 30, 34, 58] for SAT and QRAT [35] for QBF. These proof systems have been intensively studied and compared in terms of simulations, e.g., [19, 39, 40]. While the first modelling aspect needs weaker proof systems close to actual solving, the second proof-logging aspect favours very strong proof systems.

In comparison to the rich and intensely researched interplay between solving and proof complexity in SAT and QBF, significantly less is known in this regard for #SAT. In the past five years, *three different #SAT proof systems* have been introduced. These systems are kcps (2019) [18], MICE (2022) [11, 28], and CPOG (2023) [15]. These are the only #SAT proof systems so far.

The three proof systems are conceptually quite different: while kcps and CPOG are both static proof systems building on circuit classes used in knowledge compilation [24, 25] on which model counting is efficient, MICE is a rule-based proof system using three simple rules to compute counts for successively more complex formulas. The historically first system kcps was inspired by #SAT solving using knowledge compilation techniques. Both subsequent systems MICE and CPOG were designed with a view towards certifying different #SAT solving approaches.

In contrast to the rich proof complexity results for SAT and QBF, almost nothing is known theoretically for the three #SAT proof systems. Only for MICE, an exponential proof size lower bound was shown last year [11], while the relations between the three systems in terms of simulations are open.

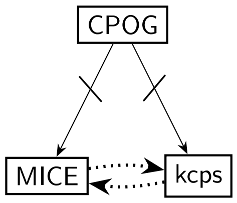


Figure 1 Simulation order of CPOG, MICE and kcps. A solid crossed edge from A to B indicates that A p-simulates B and A is exponentially stronger than B. Dotted lines indicate incomparability.

1.1 Our contributions

We perform a proof-complexity analysis of the three proof systems kcps, MICE and CPOG and completely determine their relative strength in terms of simulations and separations, leading to the picture in Figure 1. In more detail, our findings can be summarised as follows.

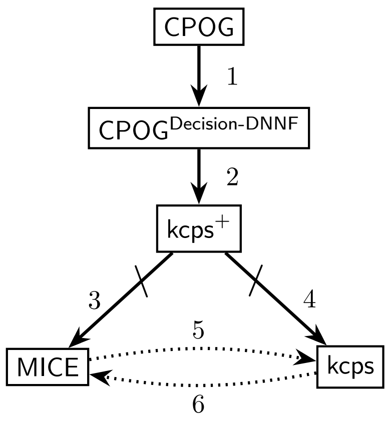
A. Simulations between #SAT proof systems. We formally compare the three #SAT proof systems in terms of *simulations* and show that CPOG p-simulates kcps and MICE, i.e. both kcps and MICE proofs can be efficiently translated into CPOG proofs.

Rather than showing the two simulations directly, we consider two intermediate proof systems kcps^+ and $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$. The first of these was already suggested in [18] as a natural extension of kcps, while $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ is newly introduced as a restriction of CPOG. The system CPOG uses POGs (partitioned-operation graphs) as the underlying circuit class, which in $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ is restricted to Decision-DNNFs: the circuit class on which kcps and kcps^+ are based. Representing a CNF by any of these circuit models allows efficient counting. Yet, the proofs need to contain additional information as verifying the equivalence of a CNF to a circuit in these models is non-trivial. While the two additional systems simplify our analysis, we believe they are also natural and of independent interest for further research (cf. the discussion in the conclusion).

For these five proof systems, we determine the simulation order as depicted in Figure 2, refining Figure 1 and including pointers to the results. While the simulations of kcps by kcps^+ and of $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ by CPOG follow almost by definition, the simulations of MICE by kcps^+ and kcps^+ by $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ are more involved – in particular the first one – as they connect conceptually quite different proof formats. The proof systems kcps, kcps^+, $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ and CPOG are all *static* as they are based on circuits equipped with additional information. In contrast, MICE is *rule-based* without any explicit connection to circuits.[1]

B. Exponential separations between #SAT proof systems. As our second main result we establish exponential separations between MICE and kcps in both directions. This entails exhibiting suitable CNF families that have short proofs in MICE, while requiring short kcps proofs, and vice versa. As a consequence, both systems are incomparable and at the same time exponentially weaker than kcps^+ and CPOG, thus resulting in the situation depicted in Figure 1.

[1] However, it was noted already in [12] that from a MICE proof a Decision-DNNF for the CNF can be extracted. This does not, however, entail a simulation of MICE by kcps (which are based on Decision-DNNFs), and in fact such a simulation *fails* as implied by our separation results in B.



Edge	Simulation	Separation
1	Observation 4.5	open
2	Theorem 4.3*	open
3	Theorem 4.1	Corollary 5.14
4	Observation 4.2*	Corollary 5.14
5	(not possible)	Corollary 5.13
6	(not possible)	Corollary 5.4

Figure 2 Detailed simulation order of #SAT proof systems. A solid edge from A to B indicates that A p-simulates B. If the edge is crossed, A is also exponentially separated from B. A dotted edge from A to B indicates that A is exponentially separated from B. All the simulations of this paper require only logarithmic space; those highlighted with "$*$" only need linear time.

Technically, we obtain one direction (kcps does not simulate MICE) by showing a tight characterisation of kcps proof size by regular resolution size on unsatisfiable formulas. A similar characterisation of MICE by full resolution was shown in [11]. As regular resolution is known to be exponentially weaker than resolution [3, 57], the separation follows.

For the other direction (MICE does not simulate kcps) we use a variant of the pebbling formulas, prominent in propositional proof complexity [8, 14]. While the small Decision-DNNFs and short kcps proofs are relatively easy to obtain, the hardness argument for MICE is technically more involved (Theorem 5.7).

The first separation positively answers an open question posed by Capelli [18] to find CNFs with polynomial-size Decision-DNNFs (these can always be extracted from short MICE proofs [12]), but no small kcps proofs. The second separation implies that we cannot efficiently transform Decision-DNNFs into MICE proofs.

1.2 Related work

For decision proof systems, there are extensive studies on simulation and separation, see, e.g., [39, 40]. For recently defined proof systems for propositional model counting, this has been open. To the best of our knowledge, this paper is the first work in this direction for model counting proof systems. However, existing approaches have been studied empirically [15,18,28]. Indeed, there is a list [27] of practical exact model counting systems, which are based on different techniques. Among these are component caching [52], treewidth [42], knowledge compilation, e.g., d4 [44], c2d [24], dsharp [46], as well as hybrid approaches [33, 42]. Some theoretical results are presented in [17] which predates the introduction of formal proof systems for #SAT. There are also clausal proof systems enriched with XOR reasoning [47]. Very recently, first steps on proof systems for approximate counting [2] have been presented.

1.3 Organisation

The remainder of this paper is organised as follows. After reviewing some standard notions from propositional logic and proof systems in Section 2, we provide formal definitions of the existing proof systems for #SAT in Section 3. We show the simulations from Figure 2 in Section 4. The separations are provided in Section 5. We conclude in Section 6 with a discussion on practical and theoretical implications.

We highlight that though we believe our results bear practical relevance, this paper performs a purely theoretical proof-complexity investigation.

2 Preliminaries

We briefly provide formal notions from propositional logic and proof systems. For more detailed information, we refer to [1, 41]. For an integer n, we set $[n] := \{1, 2, \ldots, n\}$.

Propositional formulas. A *literal* l is a variable z or its negation $\overline{z}$, and we write $\mathsf{var}(l) := z$. A *clause* is a disjunction of literals, a *conjunctive normal form (CNF)* is a conjunction of clauses. Often, we write clauses as sets of literals and formulas as sets of clauses. We assume that *propositional formulas* are given in CNF. We can efficiently transform any formula into a CNF using Tseitin transformations [54]. For a formula F, $\mathsf{vars}(F)$ denotes the set of all variables in F. If $C \in F$ is a clause and $V \subseteq \mathsf{vars}(F)$ is a set of variables, we define $C|_V = \{l \in C \mid \mathsf{var}(l) \in V\}$ and $F|_V$ denotes the formula F with every clause C replaced by $C|_V$.

Given a set V of variables, a (partial) assignment is a (partial) function $\alpha : V \to \{0, 1\}$ that maps variables to Boolean values. We write $\langle V \rangle$ for the set of all $2^{|V|}$ complete assignments to V. For a (partial) assignment α, $F[\alpha]$ denotes the formula where we replace all occurrences of variables x with $\alpha(x)$. If $F[\alpha] = 1$, we say α *satisfies* F and write $\alpha \models F$. We say that α *falsifies* F if $F[\alpha] = 0$ and write $\alpha \not\models F$.

Occasionally, we interpret an assignment as a CNF consisting of precisely those unit clauses that specify the assignment. Therefore, the set operations are well defined for formulas and assignments. We say that two assignments are *consistent* if they agree on their intersection.

A formula F is *satisfiable* if there exists an assignment $\alpha \in \langle \mathsf{vars}(F) \rangle$ such that $\alpha \models F$ and is *unsatisfiable* if there exists no such assignment. For a formula φ, $\mathsf{Mod}(\varphi) := \{\alpha \in \langle \mathsf{vars}(\varphi) \rangle \mid \alpha \models \varphi\}$ is the set of all models of φ. The *model counting problem* #SAT asks to compute $|\mathsf{Mod}(\varphi)|$ for a given formula φ. Throughout the paper, we use φ for formulas we want to count on. The SAT problem asks to decide whether a given formula is satisfiable and UNSAT whether a given formula is unsatisfiable. Moreover, we need the definition of *semantic consequence*. We write $F \models G$ if and only if for every assignment $\alpha \in \langle \mathsf{vars}(F) \rangle$, we have that $\alpha \models F$ implies $\alpha \models G$. We write $F \equiv G$ if and only if $F \models G$ and $G \models F$.

Proof systems. Following Cook and Reckhow [21], a *proof system* for a language L is a polynomial-time computable function f with range $\mathsf{rng}(f) = L$. Here, L will be chosen as either UNSAT or #SAT. If $f(w) = x$, then w is called f-proof of $x \in L$. In order to compare proof systems we need the notion of *simulations*. Let P and Q be proof systems for the same language. Then, P *p-simulates* Q if every Q-proof can be translated in polynomial time into a P-proof of the same formula. Two proof systems are *p-equivalent* if they p-simulate each other. Further, P is *exponentially separated* from Q if there is a family of formulas that have polynomial sized P-proofs while any Q-proof requires exponential size.

Resolution is arguably the most studied proof system for UNSAT. It is a *line-based* proof system with clauses as proof lines. The *resolution rule* allows to derive the clause $C \cup D$ from previously derived clauses $C \cup \{x\}$ and $D \cup \{\overline{x}\}$. We also allow the *weakening rule* that derives $C \cup D$ from a clause C. A resolution refutation of a CNF is a derivation of the empty clause $\Box$. As refutational systems, resolution with and without weakening a p-equivalent.

Further, we can interpret any proof π in a line-based proof system as a directed graph G_π, where the nodes are proof lines in π. There is an edge from proof line l_1 to l_2 if l_2 was used to derive l_1. A resolution refutation is *regular* if there is no path from the root to a leaf in G_π where a variable is resolved more than once.

3 Proof systems for #SAT

In this section, we recall the existing #SAT proof systems kcps, CPOG, and MICE and provide some intuition. In particular, we provide a concise formalisation of CPOG. Furthermore, we introduce two adapted versions of kcps and CPOG that we call kcps$^+$ and CPOG$^{\text{Decision-DNNF}}$. As kcps and CPOG heavily use concepts from knowledge compilation, we start with relevant definitions following standard texts [25, 37].

A *circuit* is a directed acyclic graph with labelled nodes that we call *gates*. We only consider circuits that have exactly one gate with indegree 0. It is called *root* and represents the circuit's output. Gates with outdegree 0 are called *leaves* and are labelled with literals or constants 0 and 1. The latter are also called 0-*gate* or 1-*gate*. Every inner gate is an AND-, OR- or NOT-gate labelled with the corresponding Boolean function. The semantics of such circuits are defined in the usual way. Additionally, we assume that AND- and OR-gates always have exactly two children.

Let D be a circuit. For gates in D we use uppercase letters such as N. We write $\mathsf{vars}(D)$ for the set of all variables occurring in leaves of D. $\mathcal{E}(D)$ denotes the Tseitin encoding [54] of D, where we use a new variable ϑ_N for every gate N. We denote the subcircuit of D with root N consisting of all descendants of N by $D(N)$.

A circuit is in *negation normal form (NNF)* if it does not contain NOT-gates. An AND-gate with children N_1 and N_2 is called *decomposable*, if $\mathsf{vars}(D(N_1)) \cap \mathsf{vars}(D(N_2)) = \emptyset$. An OR-gate with children N_1 and N_2 is called *deterministic* if there is no assignment that satisfies both $D(N_1)$ and $D(N_2)$. A DNNF [22] is an NNF where every AND-gate is decomposable. A d-DNNF [23] is a DNNF where every OR-gate is deterministic.

Since it is non-trivial to check if all OR-gates are indeed deterministic, we also consider a restricted version of d-DNNF called Decision-DNNF. In a Decision-DNNF, any OR-gate has the form $N = (N_1 \texttt{ or } N_2)$ with $N_1 = (x \texttt{ and } N_3)$ and $N_2 = (\overline{x} \texttt{ and } N_4)$ for any variable x. It is obvious that any such OR-gate is deterministic. For better readability, we write Decision-DNNFs without OR-gates but use DECISION-gates instead. We rewrite the above gate as $N = (\texttt{if } x \texttt{ then } N_3 \texttt{ else } N_4)$. Note that we can assume that the leaves of a Decision-DNNF contain only constants 0 or 1. Further, in any path from the root to a leaf, any variable can be decided at most once because of the decomposability property. We say that an assignment α *reaches* a gate N if there is a path P from the root to N such that all decisions along P are consistent with α.

3.1 Kcps: Knowledge Compilation Proof System

The system kcps is the historically first proof system for #SAT, introduced by Capelli in 2019 [18]. As the name suggests, it aims to certify solvers that apply knowledge compilation techniques. These solvers transform the input CNF into a format that can handle various queries efficiently, in particular model counting [24, 25, 44]. As in practice solvers often rely on compiling the formulas into Decision-DNNFs, kcps is based on this class of circuits.

A kcps proof of φ provides a Decision-DNNF D such that $D \equiv \varphi$. The Decision-DNNF D implicitly contains the model count of φ as we can efficiently compute it (cf. Figure 3 for an example). However, for this to be a proof in the sense of Cook-Reckhow [21], we need

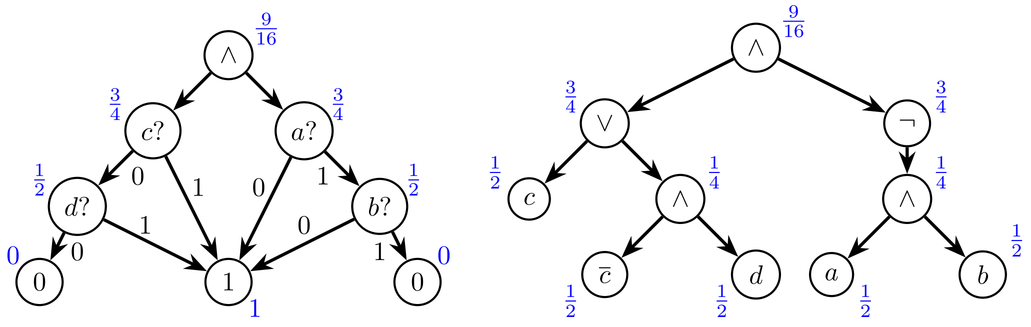


Figure 3 A Decision-DNNF (left) and a POG (right) that are equivalent to the formula $\varphi = (\overline{a} \vee \overline{b}) \wedge (c \vee d)$. The blue number at a gate N indicates the fraction of assignments that satisfy the subcircuit with root N. These numbers are computed bottom-up, i.e. 0-gates get count 0, 1-gates count 1 and gates that are labelled with a literal count $\frac{1}{2}$. DECISION-gates get the average of the two children, OR-gates the sum and AND-gates the product. Finally, to compute the model count of φ, we multiply the fraction of satisfying assignments of the whole Decision-DNNF or POG with the count of all possible assignments to $\mathsf{vars}(\varphi)$, resulting in $|\mathsf{Mod}(\varphi)| = \frac{9}{16} \cdot 2^{|\mathsf{vars}(\varphi)|} = 9$.

to verify that D and φ are indeed equivalent. The direction $D \models \varphi$ can always be checked efficiently [25] (cf. also Lemma 3.6 below for a formal argument). However, the other direction $\varphi \models D$ is a coNP-complete problem for arbitrary Decision-DNNFs [18]. Thus, we consider a restricted version of Decision-DNNFs on which checking $\varphi \models D$ becomes easy as well. For that, we review the notion of a certified Decision-DNNF [18].

▶ **Definition 3.1** (S-certified Decision-DNNF [18]). *Let S be a set of clauses. A Decision-DNNF D is called S-*certified *if every 0-gate N is labelled by a certificate $C \in S$. A clause is a certificate for N if all assignments that reach N falsify C.*

These restricted Decision-DNNFs have the property that for a formula φ, any φ-certified Decision-DNNF D satisfies $\varphi \models D$ [18]. To see this, consider the equivalent statement $\neg D \models \neg\varphi$. Let α be an assignment that falsifies D, then it reaches a 0-gate. Consequently, it has to falsify its certificate and in particular φ.

Finally, we can define the kcps proof system:

▶ **Definition 3.2** (kcps [18]). *A* kcps *proof of a CNF φ is a φ-certified Decision-DNNF D where $D \equiv \varphi$.*

Note that the model count of φ and also the equivalence between φ and D are not explicitly part of the proof as we can compute the model count efficiently from D and verify $D \equiv \varphi$ in polynomial time.

In fact, Capelli [18] proposed a generalization of kcps where the certifying clauses for the 0-gates are not necessarily clauses of the original formula φ. Instead, we use as certificates arbitrary clauses derived by resolution from φ. This results in the proof system kcps$^+$.

▶ **Definition 3.3** (kcps$^+$ [18]). *A* kcps$^+$ *proof of a CNF φ is a pair (σ, D) where*

1. *σ is a resolution derivation starting from the clauses in φ and*
2. *D is a σ-certified Decision-DNNF (i.e. all clauses labelling the 0-gates in D are derived in σ) such that $D \equiv \varphi$.*

3.2 CPOG: Certified Partitioned-Operation Graphs

In contrast to kcps, CPOG is not restricted to certified Decision-DNNFs, but uses the more flexible circuit class POG (partitioned-operation graphs). Instead of providing the original definition of POGs from [15], we equivalently define a POG as a d-DNNF with NOT-gates (alternatively, a d-DNNF can be viewed as a POG with negation applied only to variables).

Model counting is also efficient on POGs [15], and in fact POGs appear to be the largest class to which the model counting idea used for Decision-DNNFs naturally extends. However, in order to maintain efficient proof checking, a CPOG proof has to explicitly prove that P is indeed a POG and that $\varphi \equiv P$. This leads to the following definition.

▶ **Definition 3.4** (CPOG [15]). *A CPOG proof of a CNF φ is a 4-tuple $(\mathcal{E}(P), \rho, \psi, X)$ where*

1. *P is a POG with root R such that $P \equiv \varphi$ and $\mathcal{E}(P)$ is a clausal encoding of P,*
2. *ρ is a proof for $\varphi \models P$, i.e., ρ is a resolution refutation of $\mathcal{E}(P) \wedge \varphi \wedge (\overline{\vartheta_R})$,*
3. *ψ is a proof for $P \models \varphi$, i.e., ψ contains a resolution refutation of $\mathcal{E}(P) \wedge (\vartheta_R) \wedge \overline{C}$ for every clause $C \in \varphi$,*
4. *X is a set of proofs verifying that all OR-gates of P are deterministic, i.e., X is a set of resolution refutations such that for any OR-gate N, X contains a resolution refutation of $\mathcal{E}(P) \wedge (\vartheta_{N_1}) \wedge (\vartheta_{N_2})$, where N_1 and N_2 are the two child gates of N.*

Note that CPOG is originally defined on circuits with arbitrary fan-in, however we consider only the binary case which is polynomially equivalent. Additionally, the original definition uses RUP steps for the propositional proofs, which are p-equivalent to resolution. In our definition, we use resolution proofs instead.

The underlying structure of POGs in CPOG proofs is quite generic. So far, the only implementation of CPOG [15] is restricted to Decision-DNNFs instead of POGs. We capture this variant in the following definition:

▶ **Definition 3.5** ($\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$). *A $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ proof of a CNF φ is a pair $(\mathcal{E}(D), \rho)$ where*

1. *D is a Decision-DNNF with root R and $\mathcal{E}(D)$ a clausal encoding of D such that $D \equiv \varphi$,*
2. *ρ is a resolution refutation of $\varphi \wedge \mathcal{E}(D) \wedge (\overline{\vartheta_R})$.*

Note that in comparison to Definition 3.4, the last two items are missing. This is clear for item 4 as D only contains DECISION-gates instead of OR-gates. But also the proof ψ in item 3 can always be computed efficiently for Decision-DNNFs as we show in the next lemma.

▶ **Lemma 3.6.** *Let D be a Decision-DNNF with root R and encoding $\mathcal{E}(D)$. If $D \equiv \varphi$, then we can compute ψ, i.e. a resolution refutation of $\mathcal{E}(D) \wedge (\vartheta_R) \wedge \overline{C}$ for every clause $C \in \varphi$, in time $O(|D| \cdot |\varphi|)$.*

Proof sketch. Let $C \in \varphi$ be some arbitrary fixed clause. We assume that $D \equiv \varphi$. Let $L = N_1, \ldots, N_n$ be a list of all gates in D that are unsatisfiable under the partial assignment $\overline{C}$ in some topological ordering such that no gate is listed after its ancestors. Note that R is the last element in L as $D[\overline{C}] \equiv \varphi[\overline{C}]$ has to be unsatisfiable. We can show inductively that for every $i \in [n]$ we can effectively derive the unit clause $(\overline{\vartheta_{N_i}})$ from $\mathcal{E}(D) \wedge \overline{C}$. We do this by deriving $(\overline{\vartheta_{N_i}})$ from the corresponding unit clauses of its children in a constant number of resolution steps.

Since $R \in L$, we can derive $(\overline{\vartheta_R})$ efficiently. With an additional resolution step with the unit clause (ϑ_R), we derive the empty clause. In total, we can construct a resolution refutation of $\mathcal{E}(D) \wedge (\vartheta_R) \wedge \overline{C}$ of size $O(|D|)$. Since ψ contains such a refutation for every clause $C \in \varphi$, we can construct ψ with at most $O(|D| \cdot |\varphi|)$ resolution steps. ◀

Axiom. $\dfrac{}{(\emptyset, \emptyset, 1)}$ (Ax)

Composition. $\dfrac{(F, A_1, c_1) \quad \cdots \quad (F, A_n, c_n)}{(F, A, \sum_{i\in[n]} c_i)}$ (Comp)

(C-1) $\mathsf{vars}(A_1) = \cdots = \mathsf{vars}(A_n)$ and $A_i \neq A_j$ for $i \neq j$,
(C-2) $A \subseteq A_i$ for all $i \in [n]$
(C-3) there exists a resolution refutation of $A \cup F \cup \{\overline{A_i} \mid i \in [n]\}$. Such a refutation is included into the trace and is called an *absence of models statement.*

Join. $\dfrac{(F_1, A_1, c_1) \quad (F_2, A_2, c_2)}{(F_1 \cup F_2, A_1 \cup A_2, c_1 \cdot c_2)}$ (Join)

(J-1) A_1 and A_2 are consistent,
(J-2) $\mathsf{vars}(F_1) \cap \mathsf{vars}(F_2) \subseteq \mathsf{vars}(A_i)$ for $i \in \{1, 2\}$.

Extension. $\dfrac{(F_1, A_1, c_1)}{(F, A, c_1 \cdot 2^{|\mathsf{vars}(F)\setminus(\mathsf{vars}(F_1)\cup\mathsf{vars}(A))|})}$ (Ext)

(E-1) $F_1 \subseteq F$,
(E-2) $A|_{\mathsf{vars}(F_1)} = A_1$,
(E-3) A satisfies $F \setminus F_1$.

Figure 4 Inference rules for MICE [11].

3.3 MICE: Model-counting Induction by Claim Extension

The third system we need is the line-based #SAT proof system MICE, introduced with the intention to provide a proof system close to various solvers [28]. Here, we use MICE in its slightly simplified, but p-equivalent form as defined in [11].

▶ **Definition 3.7** (MICE [11, 28]). *The proof lines in* MICE *are called* claims. *A* claim *is a* 3*-tuple* (F, A, c) *consisting of a CNF* F*, a partial assignment* A *of* $\mathsf{vars}(F)$ *(called* assumption*) and a count* c*. A* MICE proof *of a CNF* φ *is a sequence of claims* $I_1, \ldots, I_k$ *that are derived with the inference rules in Figure 4 such that the final claim has the form* $(\varphi, \emptyset, c)$ *for some count* c*.*

If a MICE proof π derives the claim $(\varphi, \emptyset, c)$, then π proves that φ has exactly c models. A claim (F, A, c) is *correct* if F has exactly c models that are consistent with A. Since only correct claims can be derived in MICE [11], the count c of a correct claim (F, A, c) is uniquely determined by F and A. Thus, we sometimes omit c and use the notation (F, A) instead.

4 CPOG simulates MICE and kcps

We start our investigation by clarifying the simulation order of the #SAT proof systems introduced in Section 3 and prove that CPOG simulates MICE and kcps. We achieve this by efficiently constructing CPOG proofs from given MICE or kcps proofs. We use the systems $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ and kcps^{+} from Section 3 as convenient intermediate proof systems and show the four simulations depicted in Figure 2.

Our first simulation transforms MICE proofs into kcps^{+} proofs.

▶ **Theorem 4.1.** kcps^{+} *p-simulates* MICE.

Proof sketch. Let $\pi = I_1, \ldots, I_n$ be a MICE proof of a CNF φ with $I_k = (F_k, A_k)$ for every $k \in [n]$. Our goal is to construct a kcps$^+$ proof for φ from π. W.l.o.g. the first claim of π is $(\emptyset, \emptyset, 1)$ derived with (Ax) and all other claims are not derived with (Ax). For each $k \in [n]$, we construct a Decision-DNNF D_k that satisfies the following invariants:

(i) D_k is equivalent to $F_k[A_k]$,
(ii) D_k contains only variables from $F_k[A_k]$, and
(iii) every 0-gate N in D_k is labelled with some clause C derived from φ such that for any assignment $\alpha \in \langle \mathsf{vars}(D_k) \rangle$ that reaches N, the clause C is falsified by $\alpha \cup A_k$.

For the *base case* $k = 1$, $I_1 = (\emptyset, \emptyset, 1)$ is derived with (Ax). We set D_1 to a circuit that only contains one 1-gate. For the *induction step*, we distinguish how I_k is derived.

- *Join.* I_k is derived with (Join) from claims I_i and I_j, so we have $F_k = F_i \cup F_j$ and $A_k = A_i \cup A_j$. Per induction hypothesis, we have already derived Decision-DNNFs D_i and D_j equivalent to $F_i[A_i]$ and $F_j[A_j]$. We define D_k to be an AND-gate with the two children that are the roots of D_i and D_j.
- *Composition.* I_k is derived with (Comp) from claims $I_{i_1}, \ldots, I_{i_r}$. Per induction hypothesis, we have the corresponding circuits D_{i_j} for all $j \in [r]$. Let $V = \mathsf{vars}(A_{i_1}) \setminus \mathsf{vars}(A_k)$, keeping in mind that all assumptions A_{i_j} have the same set of variables because of (C-1). We build a complete binary decision tree T with variables in V. For every claim I_{i_j} for $j \in [r]$ there is exactly one leaf in T that is consistent with the assumption A_{i_j}. We replace this leaf with the root of the corresponding Decision-DNNF D_{i_j}. Afterwards, we replace all remaining leaves with a 0-gate. Furthermore, we remove every DECISION-gate where both decisions lead to a 0-gate as long as such gates exist. We set D_k to be the resulting circuit. For each new 0-gate we can specify a valid certificate and construct its derivation from the absence of models statement that was used for the (Comp).
- *Extension.* I_k is derived with (Ext) from I_i. Per induction hypothesis, we have already derived a Decision-DNNF D_i equivalent to $F_i[A_i]$. We set $D_k = D_i$.

This completes the induction. Since $I_n = (\varphi, \emptyset)$, D_n is a Decision-DNNF representing φ. Further, all 0-gates have valid certificates in some derivation σ. Therefore, we have constructed a valid kcps$^+$ proof $\pi' = (D_n, \sigma)$. With $|D_n| = O(n^2 \cdot |\mathsf{vars}(\varphi)|)$ and $|\sigma| = O(n^2 \cdot |\mathsf{vars}(\varphi)| \cdot |\pi|)$, we get that $|\pi'|$ is polynomial in $|\pi|$. ◀

We remark that there is a related result in [12], which shows that we can efficiently transform any MICE proof of some formula φ into a Decision-DNNF D representing φ. The theorem above strengthens this by showing that we can even derive some set σ of clauses such that all 0-gates of D are σ-certified.

Next, we observe that kcps$^+$ is indeed a generalization of kcps. This holds as we can write any kcps proof D as a kcps$^+$ proof (σ, D) where σ contains all clauses of φ.

▶ **Observation 4.2.** kcps$^+$ *p-simulates* kcps.

Now, we efficiently transform a given kcps$^+$ proof of a CNF φ into a CPOG$^{\text{Decision-DNNF}}$ proof. The choice of the Decision-DNNF D for the CPOG proof is obvious: we simply copy it from the kcps$^+$ proof. Therefore, we only have to construct a short refutation of $\varphi \models D$.

▶ **Theorem 4.3.** CPOG$^{\text{Decision-DNNF}}$ *p-simulates* kcps$^+$.

Proof. Let $\pi = (\sigma, D)$ be a kcps$^+$ proof for φ. Further, let $\mathcal{E}(D)$ be the clausal Tseitin encoding of the Decision-DNNF D with root R. For any resolution refutation ρ of $\varphi \wedge \mathcal{E}(D) \wedge (\overline{\vartheta_R})$, we obtain a valid CPOG$^{\text{Decision-DNNF}}$ proof $\pi' = (\mathcal{E}(D), \rho)$. In order to prove the theorem, we construct ρ such that $|\pi'| = O(|\pi|)$. As $|\pi'| = |\mathcal{E}(D)| + |\rho| = O(|D|) + |\rho|$ it is sufficient that $|\rho| = O(|D| + |\sigma|)$. For that, we first derive all clauses of σ in ρ.

▷ Claim 4.4. For every gate N in D, we can efficiently derive a clause $C_N = (\vartheta_N \vee C)$ from $\mathcal{E}(D) \wedge \varphi$ where C is a clause satisfying the invariant (I): *Any assignment leading to N falsifies C.*

Proof sketch. We show this by induction on the gates of D starting at the leaves. In the base case, N is a leaf. If N is a 0-gate with certificate C', we can derive C_N for $C = C'$. Otherwise, if N is an 1-gate, we can derive C_N with $C = \emptyset$. In the induction step, we use the already derived clauses C_1, C_2 corresponding to the children of N. Together with $\mathcal{E}(D)$ we can derive the clause C_N with a constant number of resolution steps. ◁

As a result, we can also derive the clause $C_R = (\vartheta_R \vee C)$ for the root R such that C satisfies (I). As there are no decisions above R, C has to be the empty clause, i.e., we have derived the unit clause $C_R = (\vartheta_R)$. By applying a resolution step with the other unit clause $(\overline{\vartheta_R})$, we refute $\varphi \wedge \mathcal{E}(D) \wedge (\overline{\vartheta_R})$ with resolution.

In total, ρ contains the derivations of σ and additionally, a constant number of resolution steps for each gate in D. Thus, the resulting resolution refutation ρ has at most size $|\sigma| + O(|D|)$ and the theorem follows. ◀

We finally show last simulation which almost follows by definition as POGs generalise Decision-DNNFs.

▶ **Observation 4.5.** CPOG *p-simulates* $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$.

Proof sketch. In order to transform a $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ proof $(\mathcal{E}(D), \rho)$ into a CPOG proof $(\mathcal{E}(P), \rho, \psi, X)$, we use that D is also a POG by setting $P = D$. We can compute the corresponding ψ efficiently as shown in Lemma 3.6. Further, it is easy to show with resolution that all Or-gates are deterministic, i.e., X can also be computed efficiently. ◀

With that, we have shown all simulations illustrated in Figure 2. Upon closer examination, all these simulations turn out to be computable with logarithmic space. Moreover, the simulations in Observation 4.2 and Theorem 4.3 can be computed in linear time.

5 MICE and kcps are incomparable

Having determined the simulation order of #SAT proof systems, we now turn to lower bounds and separations between them. We first compare MICE and kcps.

5.1 CNFs that are hard for kcps but easy for MICE

Before getting to specific lower bounds, we provide a tight characterisation of proof size on unsatisfiable formulas for kcps in terms of regular resolution.

▶ **Proposition 5.1.** *For unsatisfiable formulas,* kcps *and regular resolution are p-equivalent.*

Proof. The proof is based on [37, Theorem 18.1] stating that the minimal size of any regular resolution refutation of a formula φ equals the minimal size of any read-once branching program solving the search problem for φ. A read-once branching program for the search problem for φ is equivalent to a φ-certified Decision-DNNF D with $D \equiv \varphi$ that contains no And-gates. Thus, the result directly implies that kcps p-simulates regular resolution for unsatisfiable formulas.

For the converse simulation of kcps by regular resolution we consider an arbitrary φ-certified Decision-DNNF D with $D \equiv \varphi$ for some unsatisfiable formula φ and show that we can get rid of all And-gates:

▷ Claim 5.2. There is a φ-certified Decision-DNNF D' with $D' \equiv \varphi$ and $|D'| \leq |D|$ that contains no AND-gates.

Proof sketch. To prove this claim we present a technique to remove an AND-gate. Let N be an AND-gate of D such that all ancestors of N are DECISION-gates. Further, let N_1 and N_2 be the children of N, i.e. $D(N) \equiv D(N_1) \wedge D(N_2)$. We can argue that not both $D(N_1)$ and $D(N_2)$ are satisfiable as this would lead to a satisfying assignment of φ.

Therefore, we can assume w.l.o.g. that $D(N_1)$ is unsatisfiable, i.e. $D(N) \equiv D(N_1) \wedge D(N_2) \equiv D(N_1)$. Thus, we can decrease the number of AND-gates of D by 1 by replacing gate N with N_1. This can never increase the set of assignments that reach a particular gate, and therefore leaves all certificates intact. In this way, we can remove every AND-gate one by one without increasing the size of D. ◁

By using this claim, we convert D to some D' without AND-gates, apply the result from [37, Theorem 18.1] and obtain a regular resolution refutation of size at most $|D|$. ◀

Therefore, any lower (and upper) bound for regular resolution transfers to kcps. For regular resolution, many lower bounds are known [43], and in particular all formulas hard for resolution such as the pigeonhole principle [32] are hard for kcps. Note that any unsatisfiable formula has a trivial Decision-DNNF. Nevertheless, all kcps proofs can be of exponential size. This answers an open question from [18].

A similar proof size characterisation on unsatisfiable formulas is known for MICE, in this case in terms of full (i.e. unrestricted) resolution [11].

▶ **Proposition 5.3** ([11]). *For unsatisfiable formulas,* MICE *and resolution are p-equivalent.*

As there are CNF families exponentially separating regular and full resolution [3, 57], Propositions 5.1 and 5.3 yield:

▶ **Corollary 5.4.** MICE *is exponentially separated from* kcps.

While this separation is on unsatisfiable formulas, we can easily extend it to satisfiable CNFs as well. For this, let φ be an unsatisfiable formula that separates resolution from regular resolution. For some fresh variable $a \notin \mathsf{vars}(\varphi)$, we define $\varphi' = \{(C \vee a) \mid C \in \varphi\}$. Then, φ' has $2^{|\mathsf{vars}(\varphi)|}$ models and still separates MICE from kcps.

Firstly, φ' is still easy for MICE. We derive the claim $(\varphi', \{\overline{a}\})$ with (Comp), the absence of models statement is short since $\varphi'[\overline{a}] = \varphi$ has a short resolution refutation. Further, we derive $(\varphi', \{a\})$ with (Ext) and finally apply (Comp) to these two claims, resulting in $(\varphi', \emptyset)$.

Secondly, we argue that the hardness of φ for kcps implies the hardness of φ'. For the contrapositive, let $D \equiv \varphi'$ be a φ'-certified Decision-DNNF. Then, $D[\overline{a}] \equiv \varphi$ is a φ-certified Decision-DNNF of size at most $|D|$, i.e. φ has a kcps proof of analogous size.

5.2 CNFs that are hard for MICE but easy for kcps

Next, we show that MICE cannot simulate kcps. For that, we use a variant of the pebbling formulas on pyramidal graphs. For a given size $n \in \mathbb{N}$, the pyramidal graph (cf. Figure 5) has $m := \frac{n(n+1)}{2}$ nodes: a node $P_{i,j}$ for each $1 \leq j \leq i \leq n$. For each $i < n$, there are edges from $P_{i+1,j}$ and $P_{i+1,j+1}$ to $P_{i,j}$. The variable i is called the *row* of the node, and j is called the *column*. When comparing rows, we talk about *greater* or *smaller* rows. The nodes in row n are called sources, and the node in row 1 is called the sink.

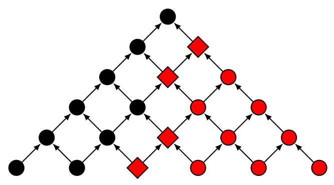


Figure 5 Pyramidal graph for PEB_6, depicting the situation in the proof of Theorem 5.7. For a fixed claim in a MICE proof of PEB_6, the red nodes are *active*, i.e. they correspond to variables that occur in the formula F. The diamond-shaped nodes form the *boundary* of this claim as they have neighbours that are not active.

We start with some intuition for the pebbling formulas PEB_n. They have two variables $w_{i,j}$ and $b_{i,j}$ for each node $P_{i,j}$. $w_{i,j}$ represents a white pebble being placed on that node, while $b_{i,j}$ represents a black pebble. The formula requires each source node to contain a pebble. Every other node needs to contain a pebble if and only if both its parent nodes contain a pebble. No node can simultaneously contain a black and a white pebble.

▶ **Definition 5.5.** *Let n be an integer. The formula PEB_n has variables $w_{i,j}$ and $b_{i,j}$ for every $i, j \in [n]$ with $j \leq i$. PEB_n is a CNF defined as follows:*

- *For every $i, j \in [n-1], j \leq i$ the formula requires that*

$$(w_{i,j} \vee b_{i,j}) \leftrightarrow ((w_{i+1,j} \vee b_{i+1,j}) \wedge (w_{i+1,j+1} \vee b_{i+1,j+1})).$$

  This is expressed using the clauses

$$\begin{aligned} C^1_{i,j} &= \overline{w_{i+1,j}} \vee \overline{w_{i+1,j+1}} \vee w_{i,j} \vee b_{i,j} & C^2_{i,j} &= \overline{w_{i+1,j}} \vee \overline{b_{i+1,j+1}} \vee w_{i,j} \vee b_{i,j} \\ C^3_{i,j} &= \overline{b_{i+1,j}} \vee \overline{w_{i+1,j+1}} \vee w_{i,j} \vee b_{i,j} & C^4_{i,j} &= \overline{b_{i+1,j}} \vee \overline{b_{i+1,j+1}} \vee w_{i,j} \vee b_{i,j} \\ C^5_{i,j} &= w_{i+1,j} \vee b_{i+1,j} \vee \overline{w_{i,j}} & C^6_{i,j} &= w_{i+1,j} \vee b_{i+1,j} \vee \overline{b_{i,j}} \\ C^7_{i,j} &= w_{i+1,j+1} \vee b_{i+1,j+1} \vee \overline{w_{i,j}} & C^8_{i,j} &= w_{i+1,j+1} \vee b_{i+1,j+1} \vee \overline{b_{i,j}}. \end{aligned}$$

- *For every $j \in [n]$ there is a clause $w_{nj} \vee b_{nj}$.*
- *For every $i, j \in [n], j \leq i$ there is a clause $C^9_{i,j} = \overline{b_{i,j}} \vee \overline{w_{i,j}}$.*

Note that the commonly used pebbling formulas require the sink node $P_{1,1}$ to contain no pebbles, making the formula unsatisfiable. We omit this requirement and obtain a formula that is satisfied if and only if each node contains exactly one pebble. It has 2^m models where m is the number of nodes. Two nodes are called *adjacent* if there is an edge between them in the pebbling graph.

To separate kcps from MICE with PEB_n, we show that there are polynomial-sized proofs in kcps while any MICE proof requires exponential size. We start with the upper bound.

▶ **Proposition 5.6.** *There is a* kcps *proof of PEB_n of size $O(|PEB_n|)$.*

Proof. We iteratively construct a PEB_n-certified Decision-DNNF D with $D \equiv \mathsf{PEB}_n$. For each node $P_{i,j}$, we construct a partial Decision-DNNF with root $N_{i,j}$ that handles the case $\{w_{i,j} = 0, b_{i,j} = 0\}$. This means that in order to obtain a valid Decision-DNNF, all paths to $N_{i,j}$ must include these two decisions. We also make sure that descendants of $N_{i,j}$ only decide variables of nodes in rows greater than i.

We begin constructing the $N_{i,j}$ for greater rows, starting with $i = n$, and continue to smaller rows. For $i = n$, $N_{i,j}$ is simply a 0-gate labelled with the clause $w_{n,j} \vee b_{n,j}$, which will be falsified by the assumption $\{w_{i,j} = 0, b_{i,j} = 0\}$.

$$\begin{tikzcd}
 & & 0\ (C^1_{i,j}) & & \\
 & w_{i+1,j+1}? \arrow[ur, "1"] \arrow[dr, "0"'] & & 0\ (C^2_{i,j}) & \\
 & & b_{i+1,j+1}? \arrow[ur, "1"] \arrow[dr, "0"'] & & \\
N_{i,j} : w_{i+1,j}? \arrow[uur, "1"] \arrow[dddr, "0"'] & & & (N_{i+1,j+1}) & \\
 & & & 0\ (C^3_{i,j}) & \\
 & & w_{i+1,j+1}? \arrow[ur, "1"] \arrow[dr, "0"'] & & 0\ (C^4_{i,j}) \\
 & b_{i+1,j}? \arrow[ur, "1"] \arrow[dr, "0"'] & & b_{i+1,j+1}? \arrow[ur, "1"] \arrow[dr, "0"'] & \\
 & & (N_{i+1,j}) & & (N_{i+1,j+1})
\end{tikzcd}$$

Figure 6 Fragment for the Decision-DNNF in Proposition 5.6. The 0-gates are certified with clauses $C_{i,j}$ from Definition 5.5.

For $i < n$, we add $N_{i,j}$ according to Figure 6. The leaves of this proof fragment are either 0-gates that are certified by some clause $C^1_{i,j}$ to $C^4_{i,j}$, or are connected to some previously constructed gate $N_{i+1,j}$ or $N_{i+1,j+1}$, after making sure that the corresponding node contains no pebbles. In this way, we can obtain a graph that contains an appropriate gate $N_{i,j}$ for every node $P_{i,j}$.

Finally, we build the complete Decision-DNNF D. For each node $P_{i,j}$, ordered from least to greatest i, we decide $w_{i,j}$ and, if it is 0, also $b_{i,j}$. If both are 0, we connect to $N_{i,j}$; if both are 1 connect to a 0-gate certified by $C^9_{i,j}$. We merge the branches of all other cases and continue with the next node. After all nodes have been handled, we finally arrive at a single 1-gate.

Because of the ordering of the nodes, each path through D can decide each variable at most once. Therefore, D is indeed a Decision-DNNF. It is equivalent to φ and φ-certified. For each node $P_{i,j}$ we add at most 13 gates, and there is one additional 1-gate. In total, the number of gates is at most $13m + 1 = O(|\mathsf{PEB}_n|)$. ◀

The actual lower bound for MICE is the more challenging part.

▶ **Theorem 5.7.** *PEB_n requires* MICE *proofs of size $2^{\Omega(n)}$.*

Note that all known lower bounds for MICE so far are based on formulas with large Decision-DNNF representation [11]. However, this lower bound technique does not work for PEB_n as it has polynomial-sized Decision-DNNFs (Proposition 5.6).

Proof. Let π be a MICE proof of PEB_n. For a claim $I = (F, A)$, the set of *active* nodes $\mathcal{A}(I)$ contains exactly the nodes $P_{i,j}$ where $w_{i,j} \in \mathsf{vars}(F)$ or $b_{i,j} \in \mathsf{vars}(F)$. We define the *width* of I as $w(I) = |\mathcal{A}(F)|$ and the *boundary* of I as $B(I) := \{N \in \mathcal{A}_F \mid$ there is an adjacent node $N' \notin \mathcal{A}_F\}$.

We show that a claim that considers about half of all nodes also needs to have a large boundary:

▷ Claim 5.8. Let I be a claim with width bounded by $\frac{m}{3} \leq w(I) \leq \frac{2\cdot m}{3}$. Then, $|B(I)| \geq \frac{n}{8}$.

Let $G_\pi = (V, E)$ be the representation of π as proof graph, i.e., V is the set of all claims in π and there is an edge (I_1, I_2) between two claim exactly if I_1 was used to derive I_2. Any claim $I \in V$ that is derived with (Join) has two incoming edges. For any such node, we delete the edge from the child that has the smaller width. We refer to the resulting graph as $G'_\pi = (V, E')$.

▷ Claim 5.9. For every model $\alpha \models \mathsf{PEB}_n$, there is a path π_α from the final claim to an (Ax) claim, along edges in E', such that for every every claim $(F, A) \in \pi_\alpha$, $\alpha \models A$.

In the rest of the proof, we use the π_α from Claim 5.9 and define V' to be the union of all π_α. We argue that claims in V' with a large boundary also have a large assumption:

▷ Claim 5.10. Any claim $I = (F, A) \in V'$ satisfies $|A| \geq |B(I)|$.

Next, we partition the claims of V' into two sets

$$X = \{I \in V' \mid w(I) < \frac{2}{3} \cdot m\},$$
$$Y = \{I \in V' \mid w(I) \geq \frac{2}{3} \cdot m\}.$$

Further, we define the set $S \subseteq Y$ as the set of nodes in Y that have a child in X, i.e.

$$S = \{I \in Y \mid \exists I_1 \in X : (I_1, I) \in E'\}.$$

We argue that all claims in S have large assumptions:

▷ Claim 5.11. Any claim $I = (F, A) \in S$ satisfies $|A| \geq \frac{n}{8}$.

On the other hand, every model of PEB_n corresponds to a claim in S:

▷ Claim 5.12. Let α be a model of PEB_n. Then, there is a claim $(F, A) \in S$ such that α and A are consistent.

Using Claims 5.11 and 5.12, we can finally prove the lower bound for the theorem. It is easy to observe that PEB_n has 2^m models because there are 2 satisfying assignments to the variables of each node. Let α be an assumption with at least $\frac{n}{8}$ variables and s the number of nodes with one or two variables in α. We observe that $2s \geq \frac{n}{8}$ and the number of models consistent with α is at most $1^s \cdot 2^{m-s} \leq 2^m \cdot 2^{n/16}$. Because each of 2^m models is consistent with a claim in S, S has at least $2^{n/16}$ elements. We conclude that $|\pi| \geq |V'| \geq |S| \geq 2^{n/16} = 2^{\Omega(n)}$ leading to the theorem. ◀

By combining Proposition 5.6 and Theorem 5.7, we finally obtain the separation of kcps from MICE:

▶ **Corollary 5.13.** kcps *is exponentially separated from* MICE.

While MICE and kcps are incomparable (Corollary 5.4, Corollary 5.13), kcps^+ simulates both systems (Theorem 4.1, Observation 4.2), which immediately leads to the following two separations.

▶ **Corollary 5.14.** kcps^+ *is exponentially separated from* MICE *and from* kcps.

With that, we have proven all separations from Figure 2.

6 Conclusion and future work

In this paper, we compare the strength of existing proof systems for #SAT. We mention that four of the systems we study, namely MICE, kcps^+, $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ and CPOG, include propositional resolution derivations in proofs. These resolution derivations are needed to check propositional entailment steps. We could define variants of the four mentioned proof systems by replacing all resolution proofs by proofs in a different propositional proof system P (and in the extreme case even with NP oracle calls). Close inspection of our results shows that all simulations and separations as depicted in Figure 2 will continue to hold when resolution is replaced throughout by an arbitrary proof system P that is at least as strong as resolution (or an NP oracle).

We discuss a few directions for further work. From a *practical perspective*, our simulation results imply that CPOG might indeed be a suitable choice for proof logging as it simulates all other #SAT proof systems. But also $\mathsf{CPOG}^{\mathsf{Decision\text{-}DNNF}}$ or kcps^+ could be practically sufficient for proof logging for all state-of-the-art #SAT solvers (and as of now, neither of these is known to be strictly weaker than CPOG).

In a related direction, we ask whether state-of-the-art knowledge compilers could effectively take advantage of kcps^+ by using resolution instead of strictly relying on existing input clauses for certificates. We see this especially in the light that component caching-based #SAT solvers, which can already be captured with MICE, can be directly turned into practically effective knowledge compilers [38]. Hence, one might ask whether we can design even stronger knowledge compilers. Alternatively, we may use kcps or CPOG to certify caching-based #SAT solvers by emitting a Decision-DNNF.

From a *theoretical perspective* the system kcps^+ appears quite interesting as it has an easy definition and is still strong enough to capture the different approaches of MICE and kcps. Designing a designated lower bound technique for kcps^+ appears to be an interesting problem.

References

1 Handbook of satisfiability, 2021.

2 S. Akshay, Supratik Chakraborty, and Kuldeep S. Meel. Auditable Algorithms for Approximate Model Counting. In *AAAI'24*, 2024.

3 Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory Comput.*, 3(1):81–102, 2007. doi:10.4086/toc.2007.v003a005.

4 Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *JAIR*, 40:353–373, 2011. doi:10.1613/jair.3152.

5 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *FOCS'03*, pages 340–351, 2003. doi:10.1109/SFCS.2003.1238208.

6 Teodora Baluta, Zheng Leong Chua, Kuldeep S. Meel, and Prateek Saxena. Scalable quantitative verification for deep neural networks. In *ICSE'21*, pages 312–323, 2021. doi:10.1109/ICSE43902.2021.00039.

7 Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *JAIR*, 22:319–351, 2004. doi:10.1613/jair.1410.

8 Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, 2004. doi:10.1007/s00493-004-0036-5.

9 Olaf Beyersdorff. Proof complexity of quantified Boolean logic – a survey. In *Mathematics for Computation (M4C)*, pages 353–391. World Scientific Publishing Company, Singapore, 2022.

10 Olaf Beyersdorff and Benjamin Böhm. Understanding the relative strength of QBF CDCL solvers and QBF resolution. *LMCS*, 19(2), 2023. doi:10.46298/lmcs-19(2:2)2023.

11 Olaf Beyersdorff, Tim Hoffmann, and Luc Nicolas Spachmann. Proof complexity of propositional model counting. In *SAT'23*, volume 271, pages 2:1–2:18, 2023. doi:10.4230/LIPIcs.SAT.2023.2.

12 Olaf Beyersdorff, Tim Hoffmann, and Luc Nicolas Spachmann. Proof complexity of propositional model counting. *ECCC*, pages TR24–030, 2024. URL: https://eccc.weizmann.ac.il/report/2024/030.

13 Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified Boolean formulas. In *Handbook of Satisfiability*, pages 1177–1221. IOS Press, 2021. doi:10.3233/FAIA201015.

14 Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. Exponential separations between restricted resolution and cutting planes proof systems. In *FOCS'98*, pages 638–647, 1998. doi:10.1109/SFCS.1998.743514.

15 Randal E. Bryant, Wojciech Nawrocki, Jeremy Avigad, and Marijn J. H. Heule. Certified knowledge compilation with application to verified model counting. In *SAT'23*, volume 271, pages 6:1–6:20, 2023. doi:10.4230/LIPIcs.SAT.2023.6.

16 Sam Buss and Jakob Nordström. Proof complexity and SAT solving. In *Handbook of Satisfiability*, pages 233–350. IOS Press, 2021. doi:10.3233/FAIA200990.

17 Florent Capelli. Understanding the complexity of #sat using knowledge compilation. In *LICS'17*, pages 1–10, 2017. doi:10.1109/LICS.2017.8005121.

18 Florent Capelli. Knowledge compilation languages as proof systems. In *SAT'19*, volume 11628, pages 90–99, 2019. doi:10.1007/978-3-030-24258-9_6.

19 Leroy Chew and Marijn J. H. Heule. Relating existing powerful proof systems for QBF. In *SAT'22*, volume 236, pages 10:1–10:22, 2022. doi:10.4230/LIPIcs.SAT.2022.10.

20 Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, volume 43, pages 143–152. ACM Books, 2023. doi:10.1145/3588287.3588297.

21 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *JSL*, 44(1):36–50, 1979. doi:10.2307/2273702.

22 Adnan Darwiche. Compiling knowledge into decomposable negation normal form. In *IJCAI'99*, pages 284–289, 1999. URL: http://ijcai.org/Proceedings/99-1/Papers/042.pdf.

23 Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *JANCL*, 11(1-2):11–34, 2001. doi:10.3166/jancl.11.11-34.

24 Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *AAAI'02*, pages 627–634, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-094.php.

25 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *JAIR*, 17:229–264, 2002. doi:10.1613/jair.989.

26 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In *AAAI'17*, pages 4488–4494, 2017. doi:10.1609/aaai.v31i1.11178.

27 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *JEA*, 26(1):1–26, 2021. doi:10.1145/3459080.

28 Johannes Klaus Fichte, Markus Hecher, and Valentin Roland. Proofs for propositional model counting. In *SAT'22*, volume 236, pages 30:1–30:24, 2022. doi:10.4230/LIPIcs.SAT.2022.30.

29 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM'08*, 2008.

30 Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891, 2003. doi:10.1109/DATE.2003.10008.

31 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*, volume 336, pages 993–1014. IOS Press, 2021. doi:10.3233/FAIA201009.

32 Amin Haken. The intractability of resolution. *TCS*, 39:297–308, 1985. doi:10.1016/0304-3975(85)90144-6.

33 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming High Treewidth with Abstraction, Nested Dynamic Programming, and Database Technology. In *SAT'20*, volume 12178, pages 343–360, 2020. doi:10.1007/978-3-030-51825-7_25.

34 Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *CADE'13*, pages 345–359, 2013. doi:10.1007/978-3-642-38574-2_24.

35 Marijn J. H. Heule, Martina Seidl, and Armin Biere. Solution validation and extraction for QBF preprocessing. *JAR*, 58(1):97–125, 2017. doi:10.1007/s10817-016-9390-4.

36 Mikolás Janota and Joao Marques-Silva. Expansion-based QBF solving versus Q-resolution. *TCS*, 577:25–42, 2015. doi:10.1016/j.tcs.2015.01.048.

37 Stasys Jukna. *Boolean Function Complexity – Advances and Frontiers*, volume 27. Springer, 2012. doi:10.1007/978-3-642-24508-4.

38 Rafael Kiesel and Thomas Eiter. Knowledge Compilation and More with SharpSAT-TD. In *KR'23*, pages 406–416, 2023. doi:10.24963/kr.2023/40.

39 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *IJCAR'18)*, volume 10900, pages 516–531, jul, 2018. Held as Part of the Federated Logic Conference (FloC'18). doi:10.1007/978-3-319-94205-6_34.

40 Benjamin Kiesl, Adrián Rebola-Pardo, Marijn J. H. Heule, and Armin Biere. Simulating strong practical proof systems with extended resolution. *JAR*, 64(7):1247–1267, 2020. doi:10.1007/s10817-020-09554-z.

41 Hans Kleine Büning and Theodor Lettman. *Propositional logic: deduction and algorithms.* Cambridge University Press, New York, NY, USA, 1999.

42 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In *CP'21*, pages 8:1–8:11, 2021. doi:10.4230/LIPIcs.CP.2021.8.

43 Jan Krajíček. *Proof complexity*, volume 170 of *Encyclopedia of Mathematics and Its Applications.* Cambridge University Press, 2019.

44 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DDNF compiler. In *IJCAI'17*, pages 667–673, Melbourne, VIC, Australia, 2017. doi:10.24963/ijcai.2017/93.

45 Anna L. D. Latour, Behrouz Babaki, Anton Dries, Angelika Kimmig, Guy Van den Broeck, and Siegfried Nijssen. Combining stochastic constraint optimization and probabilistic programming - from knowledge compilation to constraint solving. In *CP'17*, volume 10416, pages 495–511, 2017. doi:10.1007/978-3-319-66158-2_32.

46 Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *AI'12*, volume 7310, pages 356–361. Springer Verlag, 2012. doi:10.1007/978-3-642-30353-1_36.

47 Tobias Philipp and Adrián Rebola-Pardo. DRAT proofs for XOR reasoning. In *JELIA'16*, pages 415–429, 2016.

48 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011. doi:10.1016/j.artint.2010.10.002.

49 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996. doi:10.1016/0004-3702(94)00092-1.

50 Nathan Segerlind. The complexity of propositional proofs. *Bulletin of Symbolic Logic*, 13(4):417–481, 2007. doi:10.2178/bsl/1203350879.

51 Weijia Shi, Andy Shih, Adnan Darwiche, and Arthur Choi. On tractable representations of binary neural networks. In *KR'20*, pages 882–892, 2020. doi:10.24963/kr.2020/91.

52 Marc Thurley. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In *SAT'06*, pages 424–429, 2006. doi:10.1007/11814948_38.

53 Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991. doi:10.1137/0220053.

54 G. C. Tseitin. On the complexity of derivations in propositional calculus. In *Studies in Mathematics and Mathematical Logic, Part II*, pages 115–125. Springer, 1968.

55 Leslie G. Valiant. The complexity of computing the permanent. *TCS*, 8(2):189–201, 1979. doi:10.1016/0304-3975(79)90044-6.

56 Marc Vinyals. Hard examples for common variable decision heuristics. In *AAAI'20*, 2020.

57 Marc Vinyals, Jan Elffers, Jan Johannsen, and Jakob Nordström. Simplified and improved separations between regular and general resolution by lifting. In *SAT'20*, pages 182–200, 2020. doi:10.1007/978-3-030-51825-7_14.

58 Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT'14*, volume 8561, pages 422–429, 2014. doi:10.1007/978-3-319-09284-3_31.

59 Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. Check before you change: Preventing correlated failures in service updates. In *NSDI'20*, pages 575–589, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/zhai.

Clausal Congruence Closure

Armin Biere
University Freiburg, Germany

Katalin Fazekas
TU Wien, Austria

Mathias Fleury
University Freiburg, Germany

Nils Froleyks
Johannes Kepler University, Linz, Austria

Abstract

Many practical applications of satisfiability solving employ multiple steps to encode an original problem formulation into conjunctive normal form. Often circuits are used as intermediate representation before encoding those circuits into clausal form. These circuits however might contain redundant isomorphic sub-circuits. If blindly translated into clausal form, this redundancy is retained and increases solving time unless specific preprocessing algorithms are used. Furthermore, such redundant sub-formula structure might only emerge during solving and needs to be addressed by inprocessing. This paper presents a new approach which extracts gate information from the formula and applies congruence closure to match and eliminate redundant gates. Besides new algorithms for gate extraction, we also describe previous unpublished attempts to tackle this problem. Experiments focus on the important problem of combinational equivalence checking for hardware designs and show that our new approach yields a substantial gain in CNF solver performance.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability Solving, Congruence Closure, Structural Hashing, SAT Sweeping, Conjunctive Normal Form, Combinational Equivalence Checking, Hardware Equivalence Checking

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.6

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.11652423`
Dataset (Experimental Data: Log files, Plots, Tables): `https://doi.org/10.5281/zenodo.11658133`
Dataset (SAT Competition 2022 Benchmarks): `https://doi.org/10.5281/zenodo.11428010`
Dataset (SAT Competition 2023 Benchmarks): `https://doi.org/10.5281/zenodo.11426992`
Dataset (CNF encoded HWMCC'12 Miters): `https://doi.org/10.5281/zenodo.10823128`
Dataset (CNF encoded IWLS'22 Miters): `https://doi.org/10.5281/zenodo.10823099`

Funding This work was supported by the state of Baden-Württemberg through bwHPC, the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG, the Austrian Science Fund (FWF) under project No. T-1306, and by a gift from Intel Corporation.

Acknowledgements Finally, we want to thank the anonymous reviewers for valuable comments and useful suggestions, which definitely helped us to improve the paper considerably, particularly in the exposition of the experimental part.

1 Introduction

One of our motivations is to improve SAT solving for combinational equivalence checking of hardware circuits [30, 54, 63]. For decades combinational equivalence checking was considered the most successful application of formal verification in industry, actually before the SAT revolution started. Earlier approaches in the last century relied on binary decision diagram (BDD) technology, i.e., BDD sweeping [53], which however has been combined (if not replaced) with SAT sweeping [54] in this century. There are various commercial providers of equivalence checkers, including major electronic design automation (EDA) vendors such as Synopsys, Cadence, and Siemens, with widespread use in chip design.

Even though details about the inner workings of these EDA commercial equivalence checkers are not publicly available, simply encoding large equivalence checking problems into a monolithic SAT formula in conjunctive normal form (CNF) and then using a stand-alone solver to solve them does not scale. Therefore, we submitted monolithic equivalence checking benchmarks to the SAT Competition already in 2013 [24]. These benchmarks are regularly used in SAT competitions (for instance two of them in 2022) and some are still challenging.

It is fair to assume that commercial equivalence checkers use a hybrid approach, where the circuit structure guides incremental SAT queries to establish correspondence between internal sub-circuits, as a recursive process following the topological order of the circuit. These hybrid approaches to combinational equivalence checking have their own challenges [1, 70–72] and, in our view, are not a solved problem.

Furthermore, improving plain CNF-level SAT solving on such instances will be beneficial for hybrid approaches as well. Techniques useful for equivalence checking can have a positive impact on other applications of SAT too.

The question remains why state-of-the-art SAT solvers working on CNF need that guidance and are not able to efficiently find proofs for large equivalence checking problems, actually also called *miters* [30], even though, at the end, also those hybrid approaches just rely on the resolution proof system. While short proofs exist in theory, even for the simplest equivalence checking task of comparing two identical circuits, current state-of-the-art solvers based on the conflict-driven clause learning (CDCL) paradigm [26] fail to find short resolution proofs, as we have shown in previous work [45].

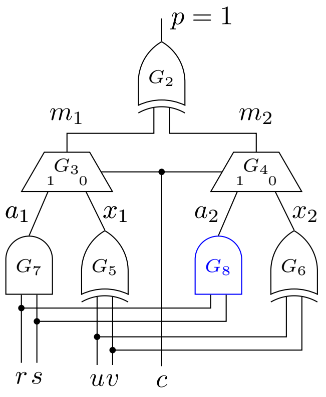
Equivalence checking of arithmetic circuits [12, 52] has similar applications and issues. In principle, algebraic techniques [34, 51, 61] can solve them, but they remain extremely challenging if given in CNF. Therefore, we consider arithmetic circuit verification out-of-scope for this study. We further focus on combinational equivalence checking leaving sequential equivalence checking, which relates to hardware model checking, to future work. Our goal is to improve CNF SAT solving for combinational (non-arithmetic) equivalence checking.

We consider *isomorphic miters*, the problem that encodes equivalence checking of two identical copies of a circuit, but also will take a look at the comparison of non-isomorphic circuits. The latter are actually the main target in industrial applications of equivalence checking, where a synthesized and optimized circuit and the original unsimplified circuit are compared. These *optimized miters* are much harder to solve.

The real cause for this failure of CDCL to solve isomorphic miters encoded into CNF is unclear, but proven empirically, as our experiments confirm. We can offer two explanation attempts though. First Yakau Novikau suggested at the Dagstuhl seminar on “The Theory and Practice of SAT Solving” in 2015 that, due to the recursive nature of equivalence checking, to learn an internal equivalence (two binary clauses) the SAT solver must fully restart in-between learning the two clauses. As a consequence, which again only empirically has been confirmed, solving miters in CNF greatly benefits from rapid restarts, i.e., restarting after each conflict. The second observation is that SAT solvers on miters even for isomorphic circuits learn rather long clauses, followed by shorter and shorter clauses until they learn some binary clauses. But then the whole process repeats, while a guided approach can focus on learning the necessary binary clauses directly.

$$\begin{aligned}
p &\underset{1}{=} 1\\
p &\underset{2}{=} m_1 \oplus m_2\\
m_1 &\underset{3}{=} c\,?\,a_1 : x_1\\
m_2 &\underset{4}{=} c\,?\,a_2 : x_2\\
x_1 &\underset{5}{=} u \oplus v\\
x_2 &\underset{6}{=} u \oplus v\\
a_1 &\underset{7}{=} r \wedge s\\
a_2 &\underset{8}{=} r \wedge s
\end{aligned}$$

(a) gates $G_1, \ldots, G_8$



(b) miter circuit

$$\begin{aligned}
&(p)_1\\
(\overline{p}\,m_1 m_2)_2\ (\overline{p}\,\overline{m}_1\overline{m}_2)_3\ (p\,\overline{m}_1 m_2)_4\ (p\,m_1\overline{m}_2)_5\\
(\overline{m}_1\overline{c}\,a_1)_6\ (\overline{m}_1 c\,x_1)_7\ (m_1\overline{c}\,\overline{a}_1)_8\ (m_1 c\,\overline{x}_1)_9\\
(\overline{m}_2\overline{c}\,a_2)_{10}\ (\overline{m}_2 c\,x_2)_{11}\ (m_2\overline{c}\,\overline{a}_2)_{12}\ (m_2 c\,\overline{x}_2)_{13}\\
(\overline{x}_1\overline{u}\,\overline{v})_{14}\ (\overline{x}_1 u\,v)_{15}\ (x_1\overline{u}\,v)_{16}\ (x_1 u\,\overline{v})_{17}\\
(\overline{x}_2\overline{u}\,\overline{v})_{18}\ (\overline{x}_2 u\,v)_{19}\ (x_2\overline{u}\,v)_{20}\ (x_2 u\,\overline{v})_{21}\\
(\overline{a}_1 r)_{22}\ (\overline{a}_1 s)_{23}\ (a_1\overline{r}\,\overline{s})_{24}\\
(\overline{a}_2 r)_{25}\ (\overline{a}_2 s)_{26}\ (a_2\overline{r}\,\overline{s})_{27}
\end{aligned}$$

(c) CNF with clauses $C_1, \ldots, C_{27}$

Figure 1 Example of an equivalence checking problem for two identical (isomorphic) circuits consisting each of one AND, XOR, and ITE (multiplexer/if-then-else) gate. The miter circuit in the middle (b) compares the output of the two circuits and assumes they are different by feeding them into another XOR gate which in turn is assumed to produce the output value 1. The equational semantics (a) is shown on the left which after Tseitin encoding [67] gives the CNF (c), e.g., the last AND gate G_8 in the second circuit is encoded by the last three clauses C_{25}, C_{26} and C_{27}.

While when working on circuits directly the gates are explicitly present, new gates can appear during solving. Our experiments on the SAT Competition 2022 shows that many SAT problems have gates, partially due to Tseitin encoding and redundant isomorphic structures. Therefore, it makes sense to have our technique on the CNF side directly: our implementation in Kissat identifies more than 180 million congruent variables.

2 Preliminaries

We assume that the reader is familiar with propositional satisfiability (SAT) and otherwise refer to [25]. In order to save space we abbreviate formulas in conjunctive normal form (CNF) by omitting operators if they are clear from the context. For instance we use $(\overline{a}r)(\overline{a}s)(a\overline{r}\,\overline{s})$ to denote the CNF $(\overline{a} \vee r) \wedge (\overline{a} \vee s) \wedge (a \vee \overline{r} \vee \overline{s})$. We identify a double negated literal with itself and denote with $|l|$ the variable v of a positive literal $l = v$ or negative literal $l = \overline{v}$.

In Fig. 1, we present an example of a combinational equivalence checking problem (miter). This is an isomorphic miter as the two circuits compared are identical. In the experiments we also consider the case where one of the circuits is an optimized version of the other, since these optimized miters are the main target in industrial applications of equivalence checking.

Hybrid approaches to equivalence checking (starting from [54] and most recently [72]) keep the two circuits alongside the CNF encoding in the SAT solver. During parsing such an isomorphic miter from a file, they will already detect all equivalences and simplify both circuits to one representation by applying "structural hashing".

This technique is also called "hash consing" in implementations of functional programming languages or "common sub-expression elimination" in compiler optimization. It is also implemented in libraries for the manipulation of binary decision diagrams (BDDs) [33] or and-inverter graphs (AIGs) [54] in the form of a "unique-table".

The basic idea of our approach is to simulate structural hashing by deriving from the CNF through resolution binary clauses of the equivalence of literals representing outputs of equivalent gates: For the two AND gates G_7 and G_8 in Fig. 1 we first derive $a_1 = a_2$, i.e., the binary clauses $(\overline{a}_1 \vee a_2)$ and $(a_1 \vee \overline{a}_2)$. Then we derive $x_1 = x_2$ for the two XOR gates G_5 and G_6. This allows us two replace the inputs a_2 and x_2 of the second ITE gate G_4 by a_1 and x_1 which in turn yields $m_1 = m_2$. Substituting m_2 with m_1 in the right hand side (RHS) of gate G_2 simplifies to 0, which contradicts the assumption that the outputs of the two compared circuits are different $(p = 1)$.

We show first how such a simulation is feasible starting from a CNF encoding and second how our new congruence closure approach solves isomorphic miters instantly. In the second scenario, when checking optimized miters, it is further expected that during solving often identical sub-circuits emerge. Our approach then allows to simplify the problem through inprocessing, which reduces over-all solving time, as confirmed in our experiments.

Related to clausal congruence closure is SAT sweeping. It has only been described in our solver description [21, 22] and uses the "small" SAT solver KITTEN within KISSAT to prove the equivalence of two literals. It can simulate congruence closure (if the variables are scheduled in the right order), but it is more expensive as it relies on KITTEN as SAT oracle. However, it is also stronger, because it is not limited to matching gates syntactically.

Our new implementation in KISSAT with efficient algorithms for gate extraction runs congruence closure until completion during both pre- and inprocessing, even for the largest CNFs in the SAT competition. We enable it by default without limit, in contrast to our earlier attempts to solve isomorphic miters including "lazy hyper binary resolution" [9], "tree-based look ahead" [45], "simple probing" (see next Sect. 3), "blocked-clause decomposition" [44], and "internal SAT sweeping" [21, 22], which all need to be limited or preempted.

3 Simple Probing

Simple probing is available in LINGELING since 2012 [10] motivated by the observation [45] that though hyper binary resolution (HBR) [3, 4, 42] combined with equivalent literal substitution (ELS) [2, 36, 56, 68] in theory can solve identical miters, in practice it fails to do so.

The problem with existing HBR implementations [3, 4, 42, 45] is that they are "global" and rely on complete failed literal probing, followed or interleaved with a global form of ELS. This means that all literals are probed and all binary clauses are taken into account in finding and substituting equivalent literals. For isomorphic miters, the fix-point of this process is only reached after many rounds of HBR and ELS. The main idea behind "simple probing" is to apply HBR and ELS steps only locally to avoid some unnecessary work.

Continuing with the example in Fig. 1, we resolve the 6 clauses $C_{22}, \ldots C_{27}$ of the two AND gates G_7 and G_8 through two hyper-binary resolution steps:

$$\frac{(a_1 \overline{r}\, \overline{s})_{24} \quad (\overline{a}_2 r)_{25} \quad (\overline{a}_2 s)_{26}}{(a_1 \overline{a}_2)_{28}}\ \mathrm{HBR}_1 \qquad \frac{(a_2 \overline{r}\, \overline{s})_{27} \quad (\overline{a}_1 r)_{22} \quad (\overline{a}_1 s)_{23}}{(a_2 \overline{a}_1)_{29}}\ \mathrm{HBR}_2$$

These two hyper binary resolution steps yield the equivalence $a_1 = a_2$, represented by the two resolvents, and correspond to the following two linear chains of resolution (RES) steps:

$$\frac{\dfrac{(a_1 \overline{r}\, \overline{s})_{24} \quad (\overline{a}_2 r)_{25}}{(a_1 \overline{a}_2 \overline{s})}\ \mathrm{RES} \qquad (\overline{a}_2 s)_{26}}{(a_1 \overline{a}_2)_{28}}\ \mathrm{RES} \qquad \frac{\dfrac{(a_2 \overline{r}\, \overline{s})_{27} \quad (\overline{a}_1 r)_{22}}{(a_2 \overline{a}_1 \overline{s})}\ \mathrm{RES} \qquad (\overline{a}_1 s)_{23}}{(a_2 \overline{a}_1)_{29}}\ \mathrm{RES}$$

Note that such linear resolution chains correspond to reverse-unit propagation (RUP) [41] in clausal proofs [46, 47]. Next we have to substitute (w.l.o.g.) a_2 by a_1 in the formula:

$$\frac{(\overline{m}_2 \overline{c}\, a_2)_{10} \quad (a_1 \overline{a}_2)_{28}}{(\overline{m}_2 \overline{c}\, a_1)_{30}}\ \mathrm{RES} \qquad \frac{(m_2 \overline{c}\, \overline{a}_2)_{12} \quad (a_2 \overline{a}_1)_{29}}{(m_2 \overline{c}\, \overline{a}_1)_{31}}\ \mathrm{RES}$$

This again boils down to resolution, which also explains why simple probing can produce RUP proofs [41] easily. Also C_{25}, C_{26}, and C_{27} of the AND gate G_8 of the circuit on the right should be substituted, but the result would be identical to the already existing clauses C_{22}, C_{23} and C_{24} of the equivalent gate G_7 of the circuit on the left, and should be avoided. Instead, they should just be deleted, the main feature in DRUP which extends the RUP proof system by including "deletion" information [69] to speed-up proof checking.

This forms the core of simple probing. In the implementation we use a counting argument: we find "immediate" hyper binary resolvents by counting how often a literal occurs in binary clauses which can be resolved with a given non-binary *base clause*. For the base clause C_{24}, we only consider the two binary clauses C_{25} and C_{26} as resolution candidates because we can ignore the blocked clauses C_{22} and C_{23} (as they both contain $\overline{a}_1$). The literal $\overline{a}_2$ occurs twice in them, and, since the base clause has one literal more than the occurrence count, this yields C_{28} through HBR_1. Similarly, we get C_{29} using C_{27} as base clause.

Whenever we find a new hyper-resolvent this way, without adding duplicates, we check whether its dual clause with both literals negated already exists. For instance, assume that in our example applying HBR_1 first. Then, when clause C_{29} is derived through HBR_2, as its dual (C_{28}) already exists, the equivalence $a_2 = a_1$ is derived. To substitute one literal with the other, we traverse all clauses containing the literal to substitute, apply the substitution, and delete the original clause. While checking for dual clauses only requires finding all binary clauses in which a literal occurs, the substitution step requires full occurrence lists.

The complete preprocessing algorithm in Alg. 2 needs to determine which and when clauses are (re)considered as base clauses. As clauses are eagerly removed and added in this approach, we do not want to use base clauses as scheduling objects in a working queue. Instead, we opted for our implementation in LINGELING to have literals occurring in base clause candidates as scheduling objects. Initially, all literals are candidate literals for simple probing. For each candidate, we go through all its non-binary clauses (requiring occurrence lists) and then apply the two-step procedure described above. After finding and substituting equivalence, we reschedule literals occurring in the resulting clauses.

Simple probing will solve isomorphic miters of circuits with only AND gates. Actually, after substituting the equivalence of outputs of the compared circuits, the comparison in clauses of the miter XOR gates will yield a unit clause. We would need to propagate those units to derive unsatisfiability (unless each compared circuit has only one output).

However, even though simple probing implicitly treats OR as AND gates, it does not handle other more complex gates, particularly neither XOR nor ITE gates. Actually, HBR+ELS alone cannot solve such miters with XORs and ITEs, including our example, as already observed by Heule at al. [45]. They proposed to interleave probing based HBR+ELS with saturating *ternary resolution* (TRN) [28] to simulate structural hashing for XOR and ITE gates, i.e., add all resolvents of at most length three between ternary gates.

Such ternary resolution is rather costly, particularly if run until completition. Thus it needs to be localized in combination with simple probing and also does not work for larger XOR gates with more than two inputs. Nevertheless, CADICAL [23] and LINGELING [11] both implement (non-localized) TRN but not eagerly and in a limited way.

4 Gate Extraction

Previous attempts (including simple probing) to solve CNF-encoded isomorphic miters through HBR (with ELS and TRN) essentially failed. They are orders of magnitude slower than circuit-based techniques, as already pointed out in the conclusion of [45] and again

Algorithm 2 Pseudo code of "simple probing" from LINGELING through local hyper binary resolution (HBR) and eager equivalent literal substitution (ELS): We interpret the given CNF F as a set of clauses, which in turn are sets of literals, with no duplicates. With $|r| \neq |l|$ in Line 10 we assume that the variables of r and l are different. Line 13 performs the actual ELS by replacing all occurrences of l with the representative literal r (resp. $\bar{l}$ by $\bar{r}$). In the actual implementation, we consider additional cases, e.g., we check for hyper binary resolved units when $\gamma(r) = |C|$ in Line 10.

```
   simple-probing (CNF $F$)          // by reference, i.e., $F$ updated in place
1    literals $L$ = all literals in $F$
2    candidates $\Lambda = L$
3    while $\Lambda \neq \emptyset$
4      pick and remove $l \in \Lambda$
5      for all "base" clauses $C \in F$ with $|C| > 2$ and $l \in C$
6        for all literals $k \in C$
7          counts $\gamma\colon L \to \mathbb{N}$ initialized to $\gamma \equiv 0$
8          for all binary clauses $(o \vee \bar{k}) \in F$
9            $\gamma(o)$++     // increment count of other literal $o$ by one
10         for all $r$ with $\gamma(\bar{r}) + 1 = |C|$ and $|r| \neq |l|$ and $(\bar{r} \vee l) \notin F$
11           add $(\bar{r} \vee l)$ to $F$     // HBR
12           if $(r \vee \bar{l}) \in F$        // checking for dual clause - ELS
13             substitute $l = r$ in all clauses $D \in F$ with $l$ or $\bar{l}$ in $D$
14             reschedule literals in resulting clauses by adding them to $\Lambda$
15             continue with outer while loop at Line 3
```

confirmed in our experiments. The key to obtaining a scalable algorithm is to extract "gates" from the CNF instead, also called "macros" and "(functional) definitions" in related work. This takes us halfway to the reconstruction of the original circuit, except that we do not care about the topological order, nor do we try to find global (primary) inputs or outputs.

Gate extraction goes back to [40, 58, 60] and we refer to the preprocessing chapter of the SAT handbook [26, Sect. 9.6.2] for details. These works were either limited in scope or had as goal to recover an actual circuit, including inputs and outputs as well as topologically ordering extracted gates. This is actually a difficult problem in general, as for instance XOR constraints (and inverters) are not directed, i.e., the Tseitin encoding of an XOR gate of arity n is symmetric in all variables and allows to actually extract $n + 1$ gates. Even for Tseitin-encoded AIGs, which are circuits with only AND gates (and inverters), there are problems. First, constant inputs might turn binary AND gates into unary AND gates (buffer/equivalences/inverters), which have to be ordered. Second, the same clause can be used for extracting multiple gates, which requires selecting a gate.

Recent work stays on the CNF level and uses blocked clause decomposition (BCD) instead [5, 44, 49, 50]. Note, however, that this approach does not support either in-processing or the production of proofs. A basic XOR-constraint extraction algorithm is described in [64] with the goal to enable algebraic reasoning. Gate detection has also been used extensively during SAT preprocessing to filter out resolvents in bounded variable elimination [37, 39]. In that context, it is local to the candidate variable for elimination and thus other algorithms apply. Similar gate-extraction approaches exist in richer logics (#SAT and QBF) too [55, 62].

We only syntactically extract "gates", trying to reverse the CNF encoding, e.g., from the clauses C_{22}, C_{23}, and C_{24} in the CNF of Fig. 1 we extract the "gate" (equation) $a_1 = r \wedge s$. Semantic extraction (such as [39, 62]) is much more powerful, but also much more expensive.

Algorithm 3 Basic algorithm for extracting AND gates. As in Alg. 2 correctness hinges on the assumption that F is without trivial clauses and all its clauses, as well as F, are interpreted as sets without duplicates. Thus, in the implementation, one must remove duplicated binary clauses first. Only binary clauses need to be watched, assuming base clauses can be traversed in some other way.

```
    basic-and-gate-extraction (CNF $F$)
1     resulting AND gates $A = \emptyset$
2     literals $L$ = all literals in $F$
3     for all clauses $C \in F$ with $|C| > 2$
4       marks $\mu\colon L \to \mathbb{B}$ initialized to $\mu \equiv \bot$     // implemented as bit-map
5       for all literals $r$ with $\overline{r} \in C$
6         $\mu(r) = \top$
7       for all literals $l \in C$
8         $n = 0$
9         for all binary clauses $(\overline{l} \vee r) \in F$
10          if $\mu(r)$ then $n$++
11        if $n = |C| - 1$
12          let $(l \vee \overline{r}_1 \vee \ldots \vee \overline{r}_n) = C$     // structured binding
13          add AND gate $(l = r_1 \wedge \cdots \wedge r_n)$ to $A$
14    return $A$
```

5 AND-Gate Extraction

Our *basic-and-gate-extraction* algorithm is shown in Alg. 3. For each non-binary base clause, it first marks the negation of all its literals. Then, for each literal in the clause, we traverse all binary clauses in which it occurs negatively. If the number of other marked literals in those binary clauses is one less than the size of the base clause, we have found an AND gate. However, for large formulas with millions of variables, millions of binary, and candidate clauses,[1] this algorithm is too slow to run until completion in order to solve miters.

In a failed improvement attempt, we added all binary clauses to a hash table, such that we can directly search for $(\overline{l} \vee r)$ when considering l as the left-hand side literal for all other r with $\overline{r} \in C$ instead of marking (Lines 5–10). However, it turns out that for large formulas, filling the hash-table took the same amount of time as the marking variant in Alg. 3.

Our first successful improvement counts the number of occurrences of literals in binary clauses and drops candidate clauses C where no literal has enough negative occurrences in binary clauses. Actually, iterations of the loop in Lines 7–13 can always be skipped in Alg. 3 for literals l where $\overline{l}$ occurs less than $|C|-1$ times in binary clauses. Our second improvement uses the observation that, while considering the left-hand-side (LHS) candidate l in Line 7 of that loop and traversing binary clauses $(\overline{l} \vee r)$ in Line 9, all remaining LHS candidates $l' \in C$ not yet tried still need to occur negated as one of these r.

For example, let $C = (l_1 \vee l_2 \vee l_3)$ in Line 3. Assume $\overline{l}_1$ occurs only once in binary clauses, and thus is skipped. Further, let $(\overline{l}_2 \vee r_1)$ and $(\overline{l}_2 \vee r_2)$ be the only binary clauses with $\overline{l}_2$ when iterating over $l = l_2$ in Line 7. If neither $l_3 = \overline{r}_1$ nor $l_3 = \overline{r}_2$ then l_3 is no LHS candidate as $(\overline{l}_2 \vee \overline{l}_3)$ is missing. To implement this optimization, we use two mark bits for the negation

[1] See e.g., `SAT_MS_sat_nurikabe_p16.pddl_166` from the main track of the SAT Competition 2022 with 19 million variables, 199 million binary clauses and 14 million candidate base clauses.

Algorithm 4 This is a basic algorithm for XOR-gate extraction. It uses the bit-extraction function β to determine if the bit at a given bit position is set and π to compute its parity.

```
basic-xor-gate-extraction (CNF $F$)
  resulting XOR gates $X = \emptyset$
  let $\beta : \mathbb{N} \times \mathbb{N} \to \{0,1\}$ with $\beta(i,s) = (s/2^i) \bmod 2$        // extract $i^{\text{th}}$ bit from $s$
  let $\pi : \mathbb{N} \to \{0,1\}$ with $\pi(s) = |\{i \mid \beta(i,s) = 1\}| \bmod 2$   // parity of all "bits" in $s$
  for all clauses $C = (l_0 \vee \ldots \vee l_{m-1}) \in F$ with $|C| > 2$
    for $s = 2$ to $2^m - 1$ with $\pi(s) = 0$                 // flip an even number of sign bits
      $D = \{l_i \mid \beta(i,s) = 0\} \cup \{\bar{l}_i \mid \beta(i,s) = 1\}$        // negate $l_i$ if $i^{\text{th}}$ bit set
      if $D \notin F$ continue with outer loop at Line 4     // clause missing
    for $i = 0$ to $m - 1$                                 // add $m$ XOR gates of arity $m-1$
      let $(l_i \vee k_1 \vee \ldots \vee k_{m-1}) = C$ and $l = \bar{l}_i$
      add XOR gate $(l = k_1 \oplus \cdots \oplus k_{m-1})$ to $X$
  return $X$
```

of literals in C. The first mark plays the same role as μ in Alg. 3 while the second is used to mark the negation of remaining LHS candidates. When counting occurrences of marked literals in Lines 9–10 we update the second mark bit and later only consider LHS literals which have the second bit still set.

6 XOR-Gate Extraction

As with AND-gate extraction, there is little published work on XOR extraction. It is briefly mentioned in [10] to support Gaussian elimination and a preliminary form of congruence closure in LINGELING for the SAT Challenge 2012. Both CADICAL since 2019 [13] and KISSAT since 2020 [20] use XOR-gate extraction to make bounded variable elimination more effective, as originally proposed in [37] for AND gates. Our basic algorithm in Alg. 4 follows these implementations and corresponds to a similar algorithm presented in [64].

In Lines 5–7, we check that all clauses D are present in the CNF which differ from the base clause C by negating exactly an even number of literals. If this is the case, we have found the XOR *constraint* $1 = l_0 \oplus \cdots \oplus l_{m-1}$, falsified by the same assignment which falsifies C (assigning all literals of C to false). This constraint can now be rewritten into those m XOR gates added on Line 10, by removing l_i from the right-hand-side (RHS) of the constraint and replacing its LHS with the negation l of l_i ("1" on the LHS above acts as negation).

The reason for adding all m gates is that we cannot (and do not want to) order symmetric gates, where input and output can be exchanged. Consequently, the functional dependency graph between inputs and outputs of our extracted gates becomes cyclic as soon as a single XOR constraint is extracted and covers all the gates. Being able to handle such cyclic dependencies is an important feature of congruence closure in our approach, which is not possible when gate extraction is used to reconstruct the structure of circuits [40, 60].

Note that for each XOR constraint found for a base clause C with m literals, the CNF actually needs to contain $2^{m-1} - 1$ matching D clauses, but we only extract m gates from it. So even for $m = 3$, we extract only three gates covering four clauses. Nevertheless, the basic algorithm performs redundant work, since Line 4 does not detect when C was already used as a matching D clause in a successful extraction before.

We can avoid this redundant work by considering in Line 4 only one of the clauses that encodes an XOR gate. Assume we have a strict order over variables, for instance, by using the integer encoding of variables in the DIMACS format. Then, C can be skipped in Line 4 unless either all literals of it are positive or only the largest one is negative. This amounts to the condition $l_0 = |l_0| < l_1 = |l_1| < \cdots < l_{m-2} = |l_{m-2}| < |l_{m-1}|$ on C in Line 4.

Note that the number of clauses needed to encode an XOR gate of arity n is 2^n, i.e., grows exponentially. As clauses in the encoding have size $m = n+1$, we can therefore further limit the size of the base clauses in Line 4 in Alg. 4. In practice, we did not see any need to search for XOR gates of arity larger than the run-time parameter $N_{\text{XOR}} = 4$.

Furthermore, as with AND gates, the XOR-extraction algorithm can be improved by counting occurrences of literals in clauses that can be part of the encoding of an XOR gate. Base clauses of size $m = |C|$ considered for extracting an XOR gate of arity $n = m - 1$ can be skipped if C contains a literal that has less than 2^{n-1} occurrences.

Finally, we realized that after counting the number of occurrences of literals in all clauses, some clauses end up having literals with too few occurrences in the reduced set of considered clauses and thus should not be considered anymore. Therefore, recounting might find additional clauses to skip. This process can be repeated until fix-point, but most of the reduction is achieved after two rounds of counting (the run-time parameter we are using).

For checking $D \notin F$ in Line 7, we connect all remaining clauses that can potentially be part of an XOR gate encoding through full occurrence lists. Searching for D can then be restricted to traverse the occurrence list of the literal in D with the minimum number of occurrences, as in backward-subsumption checks [37]. Using hashing instead (still a compile-time parameter) has similar negative results as for AND-gate extraction.

7 ITE-Gate Extraction

The most common type of encoded gates are AND gates, followed by XOR gates. Except for a few applications where they are frequent, such as describing BDDs, ITE gates occur much less often. However, occasionally it can be crucial to handle ITE gates efficiently. For example, for one of the hard synthesized miters that we considered in our experiments (`test02` from [72]) it gave a $1000\times$ improvement in solving time: 1.79 seconds when extracting vs. 2023.41 seconds when not extracting ITE gates (*cf.* Tab. 13, and Fig. 8 and 9).

As with AND and XOR gates we have been using a simple algorithm for ITE-gate extraction in the context of variable elimination for many years, i.e., where the variable of the LHS literal is fixed. A potential variant to extract all ITE gates in a given formula is shown in Alg. 5. To encode an ITE gate $(l = c\ ?\ t : e)$ exactly the following four ternary clauses are needed $(\bar{c} \vee \bar{l} \vee t)$, $(\bar{c} \vee l \vee \bar{t})$, $(c \vee \bar{l} \vee e)$, and $(c \vee l \vee \bar{e})$ ignoring two potential additional redundant clauses $(\bar{l} \vee t \vee e)$ and $(l \vee \bar{t} \vee \bar{e})$, which might be used to improve arc-consistency of the encoding. Observe that the first two clauses encode the conditional equivalence $c \rightarrow l = t$ and the third and fourth the conditional equivalence $\bar{c} \rightarrow l = e$.

The inner loop at Line 6 gives quadratic complexity in the number of literal occurrences, and with the check at Line 7 it looks even cubic. However, the actual goal of this algorithm is to find for a candidate condition c both a positive $(c \rightarrow l = t)$ and a matching negative conditional equality $(\bar{c} \rightarrow l = e)$, and thus to extract an ITE gate. This observation leads to the optimized algorithm in Alg. 6. It iterates over all variables, instead of clauses, and looks for positive and negative conditional equivalences E^+ and E^- for each of them. Equivalences of both sets with the same LHS are then merged to form ITE gates.

Algorithm 5 This is a basic algorithm for ITE-gate extraction. To find ITE gates with a *given* LHS literal l, as in variable elimination, the outer loop at Line 2 would only go over clauses with l.

```
   basic-ite-gate-extraction (CNF $F$)
1    resulting ITE gates $I = \emptyset$
2    for all ternary clauses $C = (l_1 \vee l_2 \vee l_3) \in F$
3      for $i = 1 \ldots 3$
4        let $(\bar{c} \vee \bar{l} \vee t) = C$ with $c = \bar{l}_i$
5        if $(\bar{c} \vee l \vee \bar{t}) \notin F$ continue with next $i$ at Line 3
6        for all ternary clauses $(c \vee \bar{l} \vee e) \in F$
7          if $(c \vee l \vee \bar{e}) \in F$
8            add ITE gate $(l = c \,?\, t : e)$ to $I$
9    return $I$
```

Algorithm 6 Fast ITE-gate extraction based on matching conditional equivalences.

```
    find-conditional-equivalences (CNF $F$, literal $c$)
1     resulting conditional equivalences $E = \emptyset$
2     for all ternary clauses $C = (\bar{c} \vee \bar{l} \vee t) \in F$
3       if $(\bar{c} \vee l \vee \bar{t}) \in F$
4         add $l = t$ to $E$
5     return $E$

    merge-conditional-equivalences (literal $c$, equivalences $E^+$, equivalences $E^-$)
6     resulting ITE gates $I = 0$
7     for all equivalences $l = t$ in $E^+$
8       for all equivalences $l = e$ in $E^-$
9         add ITE gate $(l = c \,?\, t : e)$ to $I$
10    return $I$

    fast-ite-gate-extraction (CNF $F$)
11    resulting ITE gates $I = 0$
12    for all variables $v$ in $F$
13      $E^+$ = find-conditional-equivalences ($F$, $v$)
14      $E^-$ = find-conditional-equivalences ($F$, $\bar{v}$)
15      add merge-conditional-equivalences ($v$, $E^+$, $E^-$) to $I$
16    return $I$
```

Further implementation details are as follows. Lines 2–3 of *find-conditional-equivalences* are implemented by extracting pairs of all the other literals in ternary clauses with $\bar{c}$, sorting the literals in the pair (smaller literal first), and then sorting all these conditional pairs lexicographically (positive literal smaller than negative). Those sorted pairs are split into "ranges" of positive and negative occurrences of the same variable as first literal in a pair.

Then we try for each pair of the smaller range to find the dual pair (with both literals negated) in the other range by binary search. Thus the complexity of *find-conditional-equivalences* is bounded by $\mathcal{O}(n \cdot \log n)$ where n is the number of ternary clauses with $\bar{c}$.

The nested loop in *merge-conditional-equivalences* can be implemented by first sorting the two conditional equivalence sets and following a merge-sort-style strategy, passing over both of them in increasing order of literals. It is still quadratic in the number of generated ITE gates, which is the worst-case complexity of the problem anyhow.

Finally, we can filter out (and do not watch) clauses which have literals that do not occur often enough: two literals (the condition and the LHS literal) have to occur twice positively and twice negatively, while the third literal must occur at least once in each polarity.

8 Congruence Closure

In SMT solvers [7] the congruence closure algorithm has found several applications, for example in ground theory solvers [57], or during quantifier instantiation [6]. It uses the congruence axiom to propagate and derive further equalities from a given set of equalities over first-order ground terms. For instance, given the equalities $x = y$, $u = f(x)$ and $v = f(y)$, the congruence axiom allows us to deduce $u = v$ too. This idea can be extended to functions and predicates of arbitrary arity. In contrast to structural hashing, it does not require any topological order of the variables, and thus can also be applied to cyclic functional definitions.

Extracted or rewritten gates need to be *normalized* to increase chances of matching other gates. For AND gates, the only form of normalization that can be achieved is to sort the RHS literals, assuming once again a fixed order on variables, e.g., induced by the variable order in the DIMACS file. The same idea can be applied to XOR gates, but besides sorting we can further force all the RHS literals of an XOR gate to be positive: if the number of negated RHS literals is even, their negations cancel, and we can simply drop them; if the number is odd, we also drop the negations and negate the LHS literal instead.

For an ITE gate ($l{=}c\ ?\ t : e$), a normalization strategy known from the BDD literature [29] applies. First, we ensure that the *condition literal* c is positive by using the equation $\bar{c}\ ?\ t : e \equiv c\ ?\ e : t$, if necessary. Then, we also make sure that the *then literal* t is positive, using $c\ ?\ \bar{t} : e \equiv \overline{c\ ?\ t : \bar{e}}$ and negating the LHS literal l if necessary.

After normalizing a gate, we check whether there is already an existing gate with the same operator (AND, ITE, XOR) and the same RHS literals. This check is implemented with a *hash table* using the operator and RHS literals as a key. If a gate is found with the same operator and RHS, we have derived an equivalence between the two LHS literals of the gates. This equivalence is recorded in a *union-find data-structure* [66], where every literal points to its (smaller) representative or itself.

Whenever a literal is assigned a new representative literal, we put that literal into a *queue*. Once all gates have been extracted, the propagation of these queued equivalences can be started in the main congruence closure loop (lines 13–18 in Alg. 7). In each iteration, a literal l of the queue is processed by iterating through all the gates that have l in their RHS. Each such gate is *rewritten* by replacing l (resp. $\bar{l}$) in them with its representative.

If a rewriting step results in a trivial gate, it is marked as garbage and skipped in later checks. For example, assume that literal b is dequeued in Line 13, and it is equivalent to its representative a. Then, the rewriting of the AND gate ($l = a \wedge b$) based on this equivalence results in the equivalence $l = a$. This we record and then mark the gate as garbage, without removing it from the RHS occurrence list of a.

Recording or *merging* an equivalence $l_1 = l_2$ consists of determining the representatives r_1 of l_1 and r_2 of l_2 (could be the literal itself). Assuming w.l.o.g. that $|r_1| < |r_2|$, we use r_1 as the new representative for both literals and push l_2 (the literal that is assigned a new representative) on the equivalence queue. As a last step, for proof logging, we augment the CNF with two binary clauses to capture that $l_2 \leftrightarrow r_1$ (this step is not shown in Alg. 7). Once the loop terminates, this augmented CNF is passed to a global equivalent literal substitution (ELS) procedure, which substitutes all equivalent literals in one pass over the formula.

Algorithm 7 An abstract version of our congruence closure algorithm. In the actual implementation we use a hash table to search gates in G by their RHS (in Lines 11 and 17) and interleave the loop in Lines 11–12 with gate extraction in Line 7. We further need to have fast access in Line 14 to all gates with the dequeued literal in their RHS, for which we use occurrence lists. We also do not show how derived unit clauses on this level of abstraction are handled which in our implementation are first propagated over the CNF and then used to simplify gates.

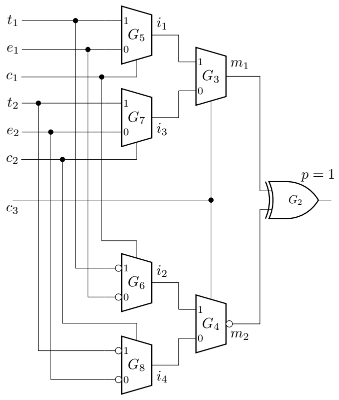
```
   *merge-literals* (CNF $F$, queue $Q$, representatives $\rho$, literals $l_1$, $l_2$)   // $F$, $Q$, $\rho$ by reference
1    $r_1 = \rho(l_1)$, $r_2 = \rho(l_2)$
2    **if** $r_1 = \bar{r}_2$ **then** $F = \bot$ **and return**   // inconsistent equivalence thus $F$ unsatisfiable
3    select $r \in \{r_1, r_2\}$ with $|r| = \mathsf{min}(|r_1|, |r_2|)$   // pick representative with smaller variable
4    update $\rho(l_1) = \rho(l_2) = r$ and $\rho(\bar{l}_1) = \rho(\bar{l}_2) = \bar{r}$
5    **if** $r \neq r_1$ **then** enqueue $l_1$ to $Q$
6    **if** $r \neq r_2$ **then** enqueue $l_2$ to $Q$
   *clausal-congruence-closure* (CNF $F$)   // by reference, i.e., $F$ updated in place
7    $G$ = *extract-gates* (F)
8    literals $L$ = all literals in $F$
9    representatives $\rho\colon L \to L$ initialized to $\rho(l) = l$
10   $Q$ = empty literal queue
11   **for all** $(l_1 = rhs_1), (l_2 = rhs_2) \in G$ with $rhs_1 = rhs_2$
12     *merge-literals* ($F$, $Q$, $\rho$, $l_1$, $l_2$)
13   **while** $F \neq \bot$ and $Q$ not empty dequeue $l$ from $Q$
14     **for** all gates $(k = rhs) \in G$ where $l$ or $\bar{l}$ occurs in $rhs$
15       use $\rho$ to rewrite $(k = rhs)$ to $(k' = rhs')$
16       remove gate $(k = rhs)$ from $G$
17       **if** $G$ contains $(k'' = rhs'')$ with $rhs' = rhs''$ **then** *merge-literals* ($F$, $Q$, $\rho$, $k'$, $k''$)
18       **else** add gate $(k' = rhs')$ to $G$
19   remove clauses $C$ from $F$ with $C \neq \rho(C) \;\wedge\; \rho(C) \in F$
20   replace $F$ with $\rho(F)$
```

Besides those (actually rather complex) ways of rewriting gates, another complication exists. It has to be taken into account when rewriting actually leads to a unit: for instance, if b in the discussed example with $(l = a \wedge b)$ has $\bar{a}$ as representative instead of a, we can derive the unit clause $\bar{l}$. In this situation, we not only propagate this new assignment through the original CNF clauses, using the existing BCP mechanism of the SAT solver, but also need to simplify all gates in which l or $\bar{l}$ occurs. Thus our loop actually consists of propagating with higher priority all literals root-level assigned to a constant through gates in which they occur on the RHS, *simplifying* them accordingly, and then with lower priority propagating equivalent literals and rewriting their gates as discussed above.

During this procedure (*cf.* Alg. 7), it might happen that an inconsistency is detected. For instance, if in the last example where $l = \bot$ is derived, the LHS l is already assigned to $\top$. Then the loop aborts and claims unsatisfiability of the formula immediately. This will in particular be the outcome when congruence closure is applied to isomorphic miters.

As already pointed out in Sect. 3, matching two isomorphic gates and substituting one LHS literal by its representative in all clauses where it occurs, necessarily results in duplicating the clauses of the representative gate. This occurs, for instance, in isomorphic miters where half of the variables vanish, but the number of clauses does not change.

$$\begin{aligned}
p &\underset{1}{=} 1\\
p &\underset{2}{=} m_1 \oplus \overline{m_2}\\
m_1 &\underset{3}{=} c_3 \;?\; i_1 : i_3\\
m_2 &\underset{4}{=} c_3 \;?\; i_2 : i_4\\
i_1 &\underset{5}{=} c_1 \;?\; t_1 : e_1\\
i_2 &\underset{6}{=} c_2 \;?\; t_2 : e_2\\
i_3 &\underset{7}{=} c_1 \;?\; \overline{t_1} : \overline{e_1}\\
i_4 &\underset{8}{=} c_2 \;?\; \overline{t_2} : \overline{e_2}
\end{aligned}$$



$$\begin{gathered}
(p)_1\\
(\overline{p}\, m_1 \overline{m}_2)_2\ (\overline{p}\, \overline{m}_1 m_2)_3\ (p\, \overline{m}_1 \overline{m}_2)_4\ (p\, m_1 m_2)_5\\
(\overline{c}_3\, \overline{m}_1\, i_1)_6\ (\overline{c}_3\, m_1\, \overline{i}_1)_7\ (c_3\, \overline{m}_1\, i_3)_8\ (c_3\, m_1\, \overline{i}_3)_9\\
(\overline{c}_3\, \overline{m}_2\, i_2)_{10}\ (\overline{c}_3\, m_2\, \overline{i}_2)_{11}\ (c_3\, \overline{m}_2\, i_4)_{12}\ (c_3\, m_2\, \overline{i}_4)_{13}\\
(\overline{c}_1\, \overline{i}_1\, t_1)_{14}\ (\overline{c}_1\, i_1\, \overline{t}_1)_{15}\ (c_1\, \overline{i}_1\, e_1)_{16}\ (c_1\, i_1\, \overline{e}_1)_{17}\\
(\overline{c}_1\, \overline{i}_2\, \overline{t}_1)_{18}\ (\overline{c}_1\, i_2\, t_1)_{19}\ (c_1\, \overline{i}_2\, \overline{e}_1)_{20}\ (c_1\, i_2\, e_1)_{21}\\
(\overline{c}_2\, \overline{i}_3\, t_2)_{22}\ (\overline{c}_2\, i_3\, \overline{t}_2)_{23}\ (c_2\, \overline{i}_3\, e_2)_{24}\ (c_2\, i_3\, \overline{e}_2)_{25}\\
(\overline{c}_2\, \overline{i}_4\, \overline{t}_2)_{26}\ (\overline{c}_2\, i_4\, t_2)_{27}\ (c_2\, \overline{i}_4\, \overline{e}_2)_{28}\ (c_2\, i_4\, e_2)_{29}
\end{gathered}$$

(a) gates $G_1, \ldots, G_8$ (b) miter circuit (c) CNF with clauses $C_1, \ldots, C_{29}$

Figure 8 An example of an optimized miter. It is comprised of an unsimplified circuit on the bottom part and its optimized variant above it. The optimized circuit simply omits the unnecessary inverters. This example also illustrates why `test02` from [72] is considerably more challenging without ITE-gate extraction. For example, to recognize easily that the output of G_5 (resp. G_7) is the negation of the output of gate G_6 (resp. G_8), the CNF encoding must maintain parts of the structure of the circuits. Extracting and normalizing ITE gates allows the congruence closure approach to realize the equivalence between the two circuits efficiently (*cf.* Fig. 9).

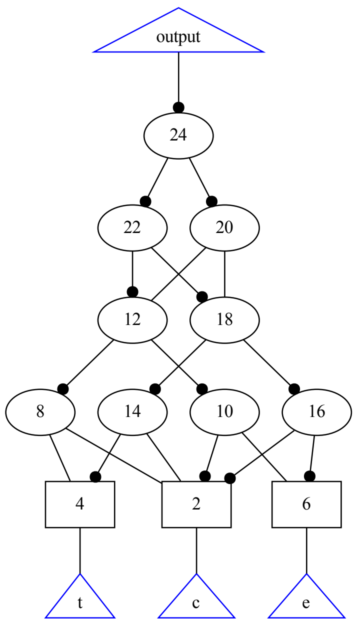
Therefore, we originally tried to eagerly delete clauses used to extract a gate as soon as it became garbage or was removed. This risks turning unsatisfiable formulas satisfiable, as clauses can be used multiple times to extract gates. Instead, we implemented a dedicated global forward subsumption algorithm (hinted at in Line 19), which targets removing identical clauses modulo equivalent literals as recorded in the union-find data structure.

9 Proofs

The algorithms for extraction and congruence closure as well as for rewriting and simplifying gates are rather involved. Therefore, we rely on generating and checking clausal proofs for correctness (i) with our internal proof checker during development and testing, as well as (ii) by producing DRUP proofs and external checking in production [48,65,69].

In principle, we just have to derive the two binary clauses for each detected equivalence. While equivalences from matched AND gates are easy to handle as they can be simulated by HBR and thus yield RUP steps as discussed in Sect. 3, equivalences from matched XOR and ITE gates require more intermediate DRUP steps as proposed in [59] for XORs or how ternary resolution is used in [45] for ITE gates and binary XORs.

Note that eager ELS during congruence closure does not need to be modelled in DRUP proofs, i.e., substituting an equivalent literal by its representative (either in clauses or in the RHS of a gate) as this is captured by propagation semantics in RUP. Proofs with hints/antecedents, such as LRAT proofs [35], would require much more effort. The internal proof checker receives the same information as the DRUP proof, but in addition we check that the clauses of the Tseitin encoding of all extracted or rewritten gates are RUP.



(a) A miter of two ITE gates in AIGER [8] format.

$$\begin{gathered}(\overline{x}_4\, x_1), (\overline{x}_4\, x_2), (x_4\, \overline{x}_1\, \overline{x}_2),\\ (\overline{x}_5\, \overline{x}_1), (\overline{x}_5\, x_3), (x_5\, x_1\, \overline{x}_3),\\ (\overline{x}_6\, \overline{x}_4), (\overline{x}_6\, \overline{x}_5), (x_6\, x_4\, x_5),\\ (\overline{x}_7\, x_1), (\overline{x}_7\, \overline{x}_2), (x_7\, \overline{x}_1\, x_2),\\ (\overline{x}_8\, \overline{x}_1), (\overline{x}_8\, \overline{x}_3), (x_8\, x_1\, x_3),\\ (\overline{x}_9\, \overline{x}_7), (\overline{x}_9\, \overline{x}_8), (x_9\, x_7\, x_8),\\ (\overline{x}_{10}\, x_6), (\overline{x}_{10}\, x_9), (x_{10}\, \overline{x}_6\, \overline{x}_9),\\ (\overline{x}_{11}\, \overline{x}_6), (\overline{x}_{11}\, \overline{x}_9), (x_{11}\, x_6\, x_9),\\ (x_{12}\, x_{10}\, x_{11}), (\overline{x}_{12}).\end{gathered}$$

(b) The ands CNF encoding of the AIG.

$$\begin{gathered}(\overline{x}_4\, \overline{x}_1\, x_3), (\overline{x}_4\, x_1\, x_2),\\ (x_4\, \overline{x}_1\, \overline{x}_3), (x_4\, x_1\, \overline{x}_2),\\ (\overline{x}_5\, \overline{x}_1\, \overline{x}_3), (\overline{x}_5\, x_1\, \overline{x}_2),\\ (x_5\, \overline{x}_1\, x_3), (x_5\, x_1\, x_2),\\ (x_6\, \overline{x}_5\, x_4), (x_6\, x_5\, \overline{x}_4), (\overline{x}_6).\end{gathered}$$

(c) The xits CNF encoding of the AIG.

Figure 9 An illustration of the difference between xits and ands CNF encodings of a given AIG. The miter applies an XOR (described by the three AND nodes A_{20}, A_{22}, and A_{24}) to compare $c\ ?\ t : e$ (AND nodes A_8, A_{10}, and A_{12}) to $\overline{c\ ?\ \overline{t} : \overline{e}}$ (AND nodes A_{14}, A_{16}, and A_{18}). The ands encoding (Fig. 9b) translates all 9 AND nodes of the AIG independently of each other, resulting in 26 clauses over 12 Boolean variables. The xits encoding (Fig. 9c), on the other hand, recognizes the ITE and XOR gates in the AIG and encodes the corresponding nodes *together* into a CNF with 11 clauses over 6 variables. While the ands encoding destroys the original ITE and XOR structures of the formula, the xits encoding maintains them. That allows our approach to recognize, extract and normalize the ITE gates efficiently and thereby the congruence closure algorithm can quickly conclude that the two ITE expressions are equivalent. This explains the efficiency of our algorithm on the `test02` miter from the IWLS'22 benchmark set (*cf.* Sect. 7 and Fig. 12).

10 Benchmarks

Our first HWMCC'12 benchmark set contains CNF encoded miters where HBR has difficulties and which had already been submitted to the SAT Competition 2013 [24]. These are miters for 341 AIGER [8] models used in the Hardware Model Checking Competition 2012. The original models are sequential and to obtain combinational miters, we simply treat latches as inputs and their next-state functions as outputs. We further used Abc [31] as synthesis tool to optimize the models (using the '`&dc2`' command). These are passed through AigMiter (from the AIGER [8] tools) to construct optimized miters, tagged opt. Isomorphic miters, tagged iso, are generated in the same way, except that optimization through Abc is skipped.

These miter circuits are then translated to CNF with a new version of AigToCnf (available in the AIGER GitHub repository and in the source code artifact [15]) which has been extended to detect XOR and ITE gates in AIGER circuits. During Tseitin encoding, we check whether an AND gate has two negated AND gates as children and actually implements an XOR or ITE gate. In this case we use a direct CNF encoding of four clauses, as for the XOR and ITE gates in Fig. 1, instead of 9 clauses for three AND gates, skipping the two child AND gates of the top AND gate. This reduces not only the number of clauses and variables but also has positive effects on running time as our experiments will show (except for the simple identical miters where there is little difference).

Therefore we have extended the original HWMCC'12 benchmark set by using this new version of AigToCnf with XOR and ITE matching too, which results in four variants of the 341 AIGER models: ands-iso, ands-opt, xits-iso, and xits-opt. We give in Fig. 8+9 an example of an optimized miter. These benchmarks are available at [17].

Our second IWLS'22 set comes from the IWLS'22 paper [72] by He-Teng Zhang, Jie-Hong R. Jiang, Alan Mishchenko, and Luca Amarù. It is an update on their DAC'21 paper [70], focusing on a hybrid approach to SAT-sweeping, i.e., using a SAT solver incrementally, taking circuit structure into account. Experiments in [72] used a subset of the benchmarks from [70]. These includes the five miters `n01`, `n04`, `n06`, and `test01` and `test02` in AIGER [8] format, provided by Alan Mishchenko. These benchmarks were considered hard for SAT sweeping, particularly for monolithic CNF-level SAT solving. Thus we consider this set of benchmarks as a litmus test for our usecase. As for HWMCC'12, our IWLS'22 CNF benchmarks come in two flavors: xits with special treatment of XOR and ITE gates during Tseitin encoding and ands without. These benchmarks are available at [16].

It turned out, confirmed by Alan Mishchenko, that the outputs of `test01` and `test02` were flipped in the generation process. This does not invalidate the SAT sweeping experiments in [70, 72] at all. However, it needs to be taken care of when encoding them into CNF with AigToCnf, by simply first negating the outputs with AigFlip. Furthermore, the other three AIGs, `n01`, `n04`, and `n06`, are not negated but have multiple outputs. Thus, we joined them by disjunction with AigOr. These tools are part of the AIGER library `https://github.com/arminbiere/aiger` and included in the source code artifact [15].

Continuing the discussion of Section 9, we not only empirically checked via fuzzing [32] that our implementation of congruence closure is sound but also that it is complete, i.e., it really solves isomorphic miters with AND, XOR, and ITE gates. To that end, we generated combinational AIGER models with our AigFuzz fuzzer, used AigMiter to produce an isomorphic miter, and then encoded it to CNF with our new version of AigToCnf, which detects XOR and ITE gates. The resulting CNF is given to Kissat using options that make sure that only congruence closure is run (to completion as always) without using any other preprocessing and not even entering the CDCL loop. Thus the CNF remains unsolved unless congruence closure alone can solve it.

11 Experiments

We follow the set-up of the main track of the SAT Competition, where each solver configuration is run on one benchmark instance in single-threaded mode. As compute platform we used the bwForCluster Helix with AMD Milan EPYC 7513 CPUs and for all experiments enforced a memory limit of 15 GB and a time limit of 5000 seconds with Runlim.

We compare our implementation of congruence closure, enabled by default in our new version of Kissat with the latest version 1.0.0 of Lingeling implementing simple probing, blocked clause decomposition (using the tools `sblitter`, followed by `mequick`, and finally using the same SAT solver Lingeling 1.0.0) [44], the winner Sbva-CaDiCaL [43] of the SAT Competition 2023, the latest version 1.9.5 of CaDiCaL [23] and MiniSat 2.2.0 [38]. We further compare against Abc [31, 72] on miter circuits. It represents the state-of-the-art [72] in hybrid SAT sweeping, but "per se" is not a solver, even though it uses SAT solvers.

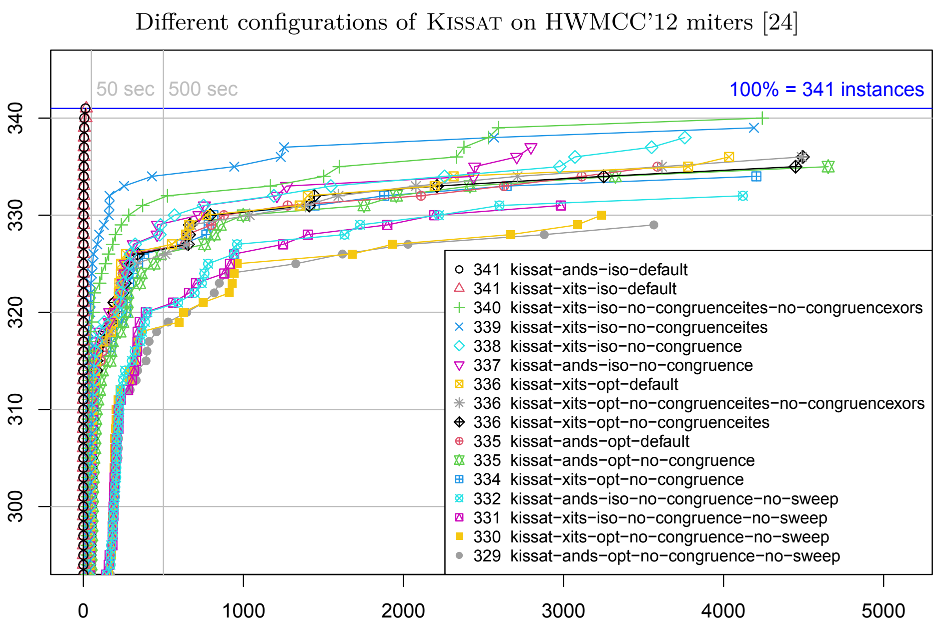


Figure 10 Comparison of variants of KISSAT with more and more relevant features disabled. The default configuration employs all of the methods described here. First, only the extraction of ITE gates is disabled (no-congruenceites), then also the extraction of XOR gates (no-congruenceites-no-congruencexors), then congruence closure is completely disabled (no-congruence), and finally even internal SAT sweeping [21, 22] is disabled (no-congruence-no-sweep). Note that for the ands encoding, no ITE nor XOR gate can be extracted anyhow and therefore disabling their extraction gives the same result as enabling them. Thus the plot shows only 6 ands variants but 10 xits variants. On this and all the following plots, the results are shown in the same way as in the annual SAT competition, e.g., a point with coordinates (1407, 333) means that 333 problems were solved in 1407 seconds.

Our results are presented as a cumulative distribution function (CDF), as in the SAT Competition since 2021, giving the number of solved problems (y-axis) within the amount of time (x-axis), i.e., the higher and the more to the left, the better. We include a horizontal line for all instances (100%). The x-axis shows time up-to the time-limit of 5000 seconds.

While adding congruence closure to KISSAT we introduced a dedicated preprocessing round, during which, after unit propagation, the first complete round of congruence closure is applied. Later, during solving, whenever probing based inprocessing is scheduled – which includes vivification, equivalent literal substitution, and other procedures – we always schedule again congruence closure elimination, but only on irredundant and binary clauses. It is also run until completion. This allows us to find additional congruent literals, as gate structure emerges after learning units, shrinking clauses, vivification, and variable elimination.

Our primary results on HWMCC'12 miters in Fig. 10 show that isomorphic miters (iso) can be solved by our new congruence closure approach kissat-$\star$-default instantly, both if we encode only AND gates directly (ands) or match them to XOR and ITE gates and then use a more elaborate Tseitin encoding (xits). Internal SAT sweeping [21, 22] implemented in KISSAT is in principle also able to find equivalences of gates. It is, however, scheduled after our faster congruence closure algorithm. For xits encoding, congruence closure takes 0.79 s on average (0 s–44.21 s) and the average percentage of total solving time is 5.4% (0.02%–26.03%). Isomorphic miters are solved by our new algorithm instantly (the vertical lines in Fig. 10).

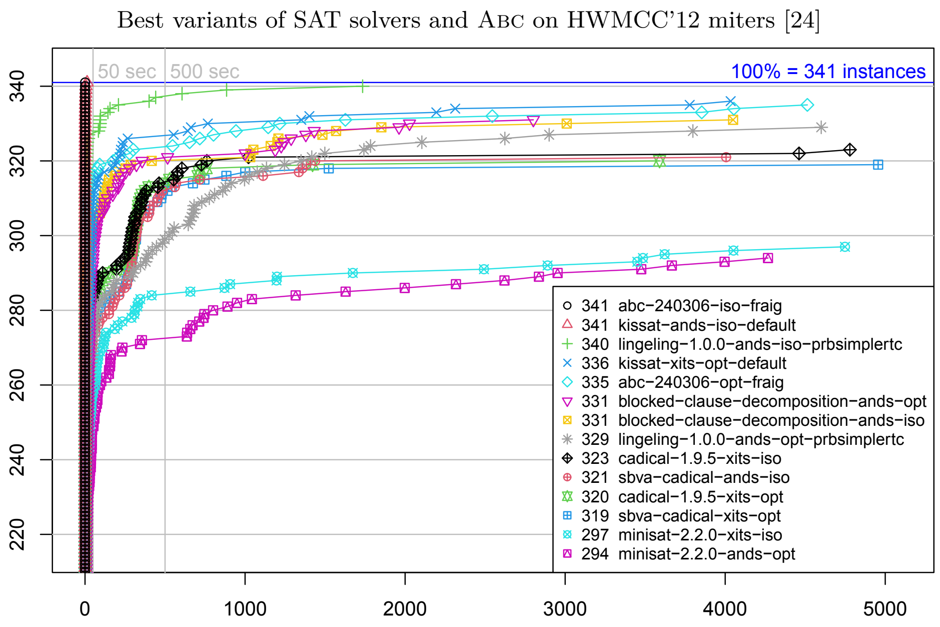


Figure 11 Number of solved HWMCC'12 miter instances at each point of time. To improve clarity and save space, we show only the best encoding variant (ands or xits) for each SAT solver. For instance our experiments revealed that for KISSAT on optimized miters (opts) the xits encoding, i.e., kissat-xits-opt-default was superior to kissat-and-opt-default with the ands encoding while for MINISAT it was the opposite (and therefore we only show minisat-2.2.0-ands-opt and not minisat-2.2.0-xits-opt).

Comparison with other solvers is shown in Fig. 11. Optimized miters (opt) are in general harder to solve, but clausal congruence closure, as enabled by default in kissat-xits-opt-default, even surpasses abc-240306-opt-frag, which represents the state-of-the-art in hybrid SAT sweeping [72], as implemented in ABC (command '`&fraig -y`'). Running simple probing in LINGELING until completion (lingeling-1.0.0-ands-iso-prbsimplertc) is the only CNF-level approach that can compete on isomorphic miters (iso), but is not competitive on optimized (opt) ones (Fig. 12, Tab. 13). Note that simple probing can not handle XOR nor ITE gates.

Fig. 12 shows the results on IWLSS'22 benchmarks. Our implementation in KISSAT (in contrast to ABC) can provide DRAT proofs [69] as standard in the SAT Competition. Actually only DRUP proofs are relevant for congruence closure, as described in Sect. 9. The results demonstrate that the overhead for proof production (proof) for KISSAT is low and proof checking (check) has comparable run-time to solving. On these 5 benchmarks our new algorithm gives substantial improvements in solving time, i.e., kissat-xits-default vs. kissat-xits-no-congruence in Tab. 13. On four of these benchmarks ABC still wins (running on the AIGER circuit model while KISSAT only gets CNF) except for `test02` where KISSAT is faster for the xits encoding. See Fig. 8+9 for an explanation.

To assess the effectiveness of congruence closure on a more general set of problems, we also evaluated our new version of KISSAT on problem instances [18, 19] from the SAT Competition 2022 and 2023. The results show that our implementation is fast enough to run to completion even when there are few or no gates to extract. On the 2022 problems our method solves 14 more instances (Fig. 14). The effect of congruence closure on the 2023 problem set is small (*cf.* Fig. 15), probably due to a large fraction of combinatorial benchmarks. Following the SAT practitioner manifesto [27] we also compare against the 2023 winner SBVA-CADICAL.

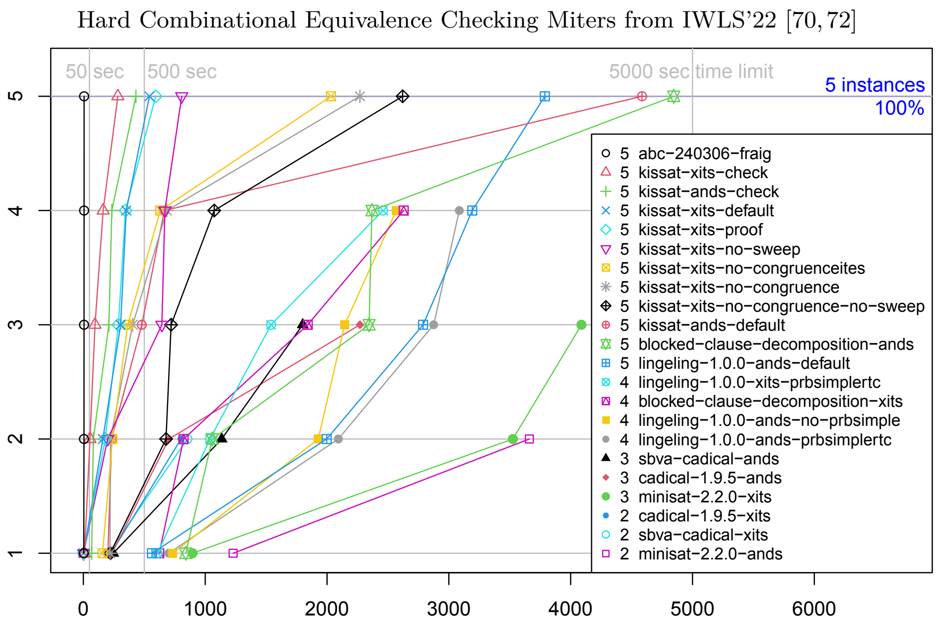


Figure 12 These miters from [70] were target of optimizations reported in [72]. They are indeed hard for monolithic SAT solving starting after Tseitin encoding. Results on benchmark `test02` are particularly interesting as kissat-xits-default took 1.79 s. to solve it, ABC 5.75 s, while disabling extraction of ITE gates in kissat-xits-no-congruenceites already needs 2032.41 s and plain AND-only Tseitin encoding in kissat-ands-default even 4585.76 s (*cf.* Tab. 13 for more detailed results).

Table 13 The actual run-time on the IWLS'22 miters from [70, 72] (*cf.* CDF in Fig. 12).

	n01	n04	n06	test01	test02
abc-240306-fraig	5.96	5.38	4.86	2.89	5.75
kissat-xits-check	95.45	162.38	282.61	54.28	9.06
kissat-ands-check	81.57	209.54	233.75	67.95	431.21
kissat-xits-default	305.60	160.01	542.18	352.15	1.79
kissat-xits-proof	287.21	179.72	593.54	345.60	2.38
kissat-xits-no-sweep	199.17	807.07	644.22	669.11	1.79
kissat-xits-no-congruenceites	238.32	157.06	631.46	363.93	2032.41
kissat-xits-no-congruence	222.25	218.73	684.94	404.48	2270.00
kissat-xits-no-congruence-no-sweep	221.25	678.17	720.29	1073.75	2620.65
kissat-ands-default	231.87	201.45	664.81	479.28	4585.76
blocked-clause-decomposition-ands	840.19	1058.28	2345.20	2368.54	4846.14
lingeling-1.0.0-ands-default	563.03	3192.09	1997.28	2788.51	3788.10
lingeling-1.0.0-xits-prbsimplertc	607.82	1039.04	1540.55	2459.75	—
blocked-clause-decomposition-xits	622.46	822.68	1841.48	2628.96	—
lingeling-1.0.0-ands-no-prbsimple	733.61	1928.03	2144.69	2568.83	—
lingeling-1.0.0-ands-prbsimplertc	700.58	3085.86	2092.79	2875.45	—
sbva-cadical-ands	244.94	1800.21	1135.28	—	—
cadical-1.9.5-ands	236.14	2270.17	701.13	—	—
minisat-2.2.0-xits	895.77	4088.40	3525.61	—	—
cadical-1.9.5-xits	227.21	—	801.69	—	—
sbva-cadical-xits	205.70	—	853.77	—	—
minisat-2.2.0-ands	1229.07	—	3660.71	—	—

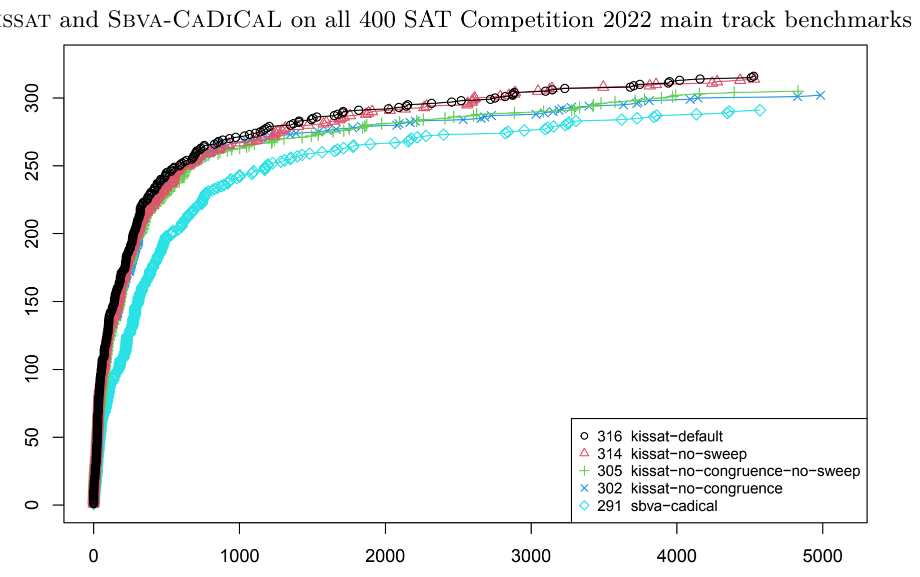


Figure 14 On the problems of the main track of the SAT Competition 2022 [18] the congruence closure algorithm is successful. In fact, all versions of Kissat are faster than Sbva-CaDiCaL. Two benchmarks `6133-sc2014` and `6s184`, reused from our HWMCC'12 isomorphic miter benchmarks submitted to the SAT Competition 2013 [24], were solved immediately by congruence closure (in 0.07 s and 0.04 s), but were also solved without congruence closure (in 37.51 s and 25.99 s). The `default` configuration of Kissat eliminated a total of 108 272 236 equivalent literals found by congruence closure among all the 400 benchmarks of the main track.

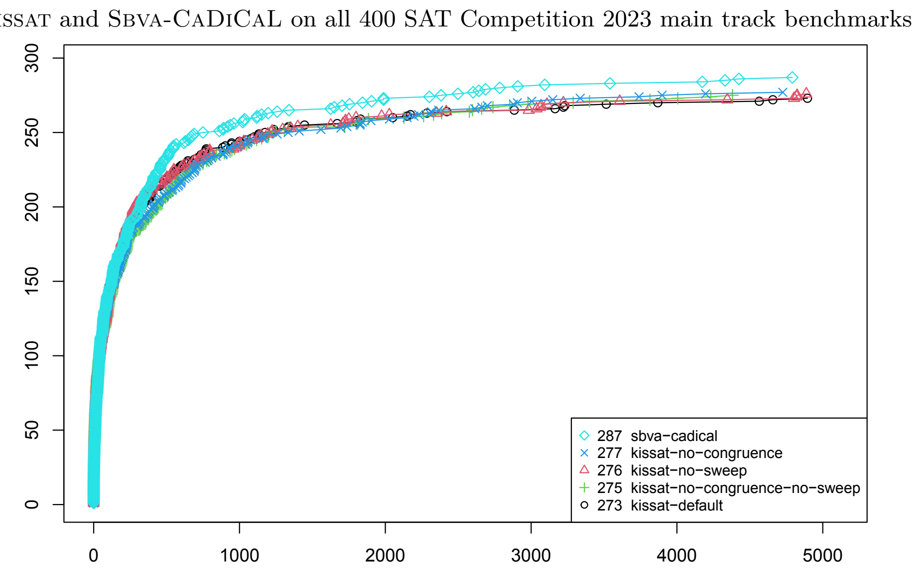


Figure 15 In the main track of the SAT Competition 2023 [19] with many hard combinatorial problems, Structured Bounded Variable Addition (SBVA) [43] in Sbva-CaDiCaL, the winner of this track, has an advantage over Kissat, because SBVA and congruence closure are orthogonal. The different variants of congruence closure are very similar here, although the `default` version spent on average 4.41% of the running time in congruence closure.

Finally, we want to investigate the average learned clause length, related to the observation in the introduction on CDCL not being able to produce short proofs. Therefore, we have rerun without congruence closure (no-congruence) but with more statistics all the isomorphic HWMCC'12 miters again (see the metrics directories in the experimental data artifact [14]) and computed the average learned clause lengths over all miters, which is 43.6 literals per learned clause for ands-iso-no-congruence, and 46.7 for xits-iso-no-congruence. Our default version of KISSAT with congruence closure solves these miters instantly through preprocessing, without the need to learn any clause, and thus we computed instead the average added clause length in the RUP proofs which is 1.88 literals for ands-iso and 2.12 for xits-iso.

Source code is available on Zenodo [15]. The HWMCC'12 benchmarks [17] and ILWS'22 benchmarks [16] are available on Zenodo too, as well as all experimental data [14].

12 Conclusion

We explored the idea of applying congruence closure to gates extracted from CNF using an inverse of the Tseitin encoding. Our new optimized extraction algorithms for AND, XOR, and ITE gates are able to run until completion within seconds on large combinational equivalence checking miters and benchmarks from the SAT competition. These gates are then used in a congruence closure algorithm to match equivalent gates and deduce equivalent literals, which can also run to completion on standard benchmarks from the SAT competition and is now enabled by default in our new version of the SAT solver KISSAT.

Our experiments show that this is the first approach in the literature to instantly solve large isomorphic CNF encoded miters. Further, it gives substantial improvements on industrially relevant optimized miters, where our CNF level approach reaches the performance or even is better than a dedicated circuit level SAT sweeping technique.

References

1 Luca G. Amarù, Felipe S. Marranghello, Eleonora Testa, Christopher Casares, Vinicius N. Possani, Jiong Luo, Patrick Vuillod, Alan Mishchenko, and Giovanni De Micheli. Sat-sweeping enhanced for logic synthesis. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020. doi:10.1109/DAC18072.2020.9218691.

2 Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979. doi:10.1016/0020-0190(79)90002-4.

3 Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 – August 1, 2002, Edmonton, Alberta, Canada*, pages 613–619. AAAI Press / The MIT Press, 2002.

4 Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355. Springer, 2003. doi:10.1007/978-3-540-24605-3_26.

5 Tomás Balyo, Andreas Fröhlich, Marijn Heule, and Armin Biere. Everything you always wanted to know about blocked sets (but were afraid to ask). 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 317–332. Springer, 2014. doi:10.1007/978-3-319-09284-3_24.

6 Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 214–230, 2017. doi:10.1007/978-3-662-54580-5_13.

7 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. doi:10.3233/FAIA201017.

8 Armin Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

9 Armin Biere. P{re,i}coSAT@SC'09. In *SAT 2009 Competitive Event Booklet*, 2009. URL: http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.

10 Armin Biere. Lingeling and friends entering the SAT Challenge 2012. In Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors, *Proc. of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*, pages 33–34. Univ.Helsinki, 2012.

11 Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In Adrian Balint, Andon Belov, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2014 – Solver and Benchmark Descriptions*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. Univ.Helsinki, 2014.

12 Armin Biere. Collection of combinational arithmetic miters submitted to the SAT Competition 2016. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*, pages 65–66. Univ.Helsinki, 2016.

13 Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. Univ.Helsinki, 2019.

14 Armin Biere. Clausal congruence closure paper logs, plots and tables, June 2024. doi:10.5281/zenodo.11658133.

15 Armin Biere. Clausal congruence closure paper source code, June 2024. doi:10.5281/zenodo.11652423.

16 Armin Biere. CNF encoded hard miters from IWLS'22 paper, March 2024. doi:10.5281/zenodo.10823099.

17 Armin Biere. CNF encoded isomorphic and optimized miters from Hardware Model Checking Competition 2012 models, March 2024. doi:10.5281/zenodo.10823128.

18 Armin Biere. SAT Competition 2022 main track benchmarks, June 2024. doi:10.5281/zenodo.11428010.

19 Armin Biere. SAT Competition 2023 main track benchmarks, June 2024. doi:10.5281/zenodo.11426992.

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

21 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. Univ.Helsinki, 2022.

22 Armin Biere, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Report Series B*, pages 10–13. Univ.Helsinki, 2021.

23 Armin Biere, Mathias Fleury, and Florian Pollitt. CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 14–15. Univ.Helsinki, 2023.

24 Armin Biere, Marijn Heule, Matti Järvisalo, and Norbert Manthey. Equivalence checking of HWMCC 2012 circuits. In Adrian Balint, Andon Belov, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2013 – Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, page 104. Univ.Helsinki, 2013.

25 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. doi:10.3233/FAIA336.

26 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. doi:10.3233/FAIA200992.

27 Armin Biere, Matti Järvisalo, Daniel Le Berre, Kuldeep S. Meel, and Stefan Mengel. The SAT Practitioner's Manifesto, September 2020. doi:10.5281/zenodo.4500928.

28 Alain Billionnet and Alain Sutter. An efficient algorithm for the 3-satisfiability problem. *Oper. Res. Lett.*, 12(1):29–36, 1992. doi:10.1016/0167-6377(92)90019-Y.

29 Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In Richard C. Smith, editor, *Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990*, pages 40–45. IEEE Computer Society Press, 1990. doi:10.1145/123186.123222.

30 Daniel Brand. Verification of large synthesized designs. In Michael R. Lightner and Jochen A. G. Jess, editors, *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 534–537. IEEE Computer Society / ACM, 1993. doi:10.1109/ICCAD.1993.580110.

31 Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010. doi:10.1007/978-3-642-14295-6_5.

32 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010. doi:10.1007/978-3-642-14186-7_6.

33 Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

34 Maciej J. Ciesielski, Cunxi Yu, Walter Brown, Duo Liu, and André Rossi. Verification of gate-level arithmetic circuits by function extraction. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 52:1–52:6. ACM, 2015. doi:10.1145/2744769.2744925.

35 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.

36 Alvaro del Val. Simplifying binary propositional theories into connected components twice as fast. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2001. `doi:10.1007/3-540-45653-8_27`.

37 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

38 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. `doi:10.1007/978-3-540-24605-3_37`.

39 Mathias Fleury and Armin Biere. Mining definitions in Kissat with Kittens. *Formal Methods Syst. Des.*, 60(3):381–404, 2022. `doi:10.1007/S10703-023-00421-2`.

40 Zhaohui Fu and Sharad Malik. Extracting logic circuit structure from conjunctive normal form descriptions. In *20th International Conference on VLSI Design (VLSI Design 2007), Sixth International Conference on Embedded Systems (ICES 2007), 6-10 January 2007, Bangalore, India*, pages 37–42. IEEE Computer Society, 2007. `doi:10.1109/VLSID.2007.81`.

41 Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2008, Fort Lauderdale, Florida, USA, January 2-4, 2008*, 2008. URL: `http://isaim2008.unl.edu/PAPERS/TechnicalProgram/ISAIM2008_0008_60a1f9b2fd607a61ec9e0feac3f438f8.pdf`.

42 Roman Gershman and Ofer Strichman. Cost-effective hyper-resolution for preprocessing CNF formulas. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 423–429. Springer, 2005. `doi:10.1007/11499107_34`.

43 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective auxiliary variables via structured reencoding. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. `doi:10.4230/LIPICS.SAT.2023.11`.

44 Marijn Heule and Armin Biere. Blocked clause decomposition. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2013. `doi:10.1007/978-3-642-45221-5_29`.

45 Marijn Heule, Matti Järvisalo, and Armin Biere. Revisiting hyper binary resolution. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2013. `doi:10.1007/978-3-642-38171-3_6`.

46 Marijn J. H. Heule. Proofs of unsatisfiability. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 635–668. IOS Press, 2021. `doi:10.3233/FAIA200998`.

47 Marijn J. H. Heule and Armin Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All (APPA)*, volume 55 of *Math. Logic and Foundations*. College Pub., 2015.

48 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. `doi:10.1007/S10817-019-09516-0`.

49 Markus Iser. *Recognition and Exploitation of Gate Structure in SAT Solving*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2020. URL: `https://nbn-resolving.org/urn:nbn:de:101:1-2020042904595660732648`.

50 Markus Iser, Felix Kutzner, and Carsten Sinz. Using gate recognition and random simulation for under-approximation and optimized branching in SAT solvers. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 1029–1036. IEEE Computer Society, 2017. `doi:10.1109/ICTAI.2017.00158`.

51 Daniela Kaufmann and Armin Biere. Improving AMulet2 for verifying multiplier circuits using SAT solving and computer algebra. *Int. J. Softw. Tools Technol. Transf.*, 25(2):133–144, 2023. `doi:10.1007/s10009-022-00688-6`.

52 Daniela Kaufmann, Manuel Kauers, Armin Biere, and David Cok. Arithmetic verification problems submitted to the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, page 49. Univ.Helsinki, 2019.

53 Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In Ellen J. Yoffa, Giovanni De Micheli, and Jan M. Rabaey, editors, *Proceedings of the 34st Conference on Design Automation, Anaheim, California, USA, Anaheim Convention Center, June 9-13, 1997*, pages 263–268. ACM Press, 1997. `doi:10.1145/266021.266090`.

54 Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 21(12):1377–1394, 2002. `doi:10.1109/TCAD.2002.804386`.

55 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artif. Intell.*, 281:103229, 2020. `doi:10.1016/j.artint.2019.103229`.

56 Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 – August 3, 2000, Austin, Texas, USA.*, pages 291–296. AAAI Press / The MIT Press, 2000.

57 Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007. `doi:10.1016/J.IC.2006.08.009`.

58 Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Recovering and exploiting structural knowledge from CNF formulas. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming – CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002. `doi:10.1007/3-540-46135-3_13`.

59 Tobias Philipp and Adrian Rebola-Pardo. DRAT proofs for XOR reasoning. In Loizos Michael and Antonis C. Kakas, editors, *Logics in Artificial Intelligence – 15th European Conference, JELIA 2016, Larnaca, Cyprus, November 9-11, 2016, Proceedings*, volume 10021 of *Lecture Notes in Computer Science*, pages 415–429, 2016. `doi:10.1007/978-3-319-48758-8_27`.

60 Jarrod A. Roy, Igor L. Markov, and Valeria Bertacco. Restoring circuit structure from SAT instances. In *Proceedings of International Workshop on Logic and Synthesis (IWLS)*, pages 663–678, 2004.

61 Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In Luca Fanucci and Jürgen Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1048–1053. IEEE, 2016. URL: `https://ieeexplore.ieee.org/document/7459464/`.

62 Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification – 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020. `doi:10.1007/978-3-030-53288-8_24`.

63 Gordon L. Smith, Ralph J. Bahnsen, and Harry Halliwell. Boolean comparison of hardware and flowcharts. *IBM J. Res. Dev.*, 26(1):106–116, 1982. doi:10.1147/RD.261.0106.

64 Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 – February 1, 2019*, pages 1592–1599. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33011592.

65 Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in cakeml. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2021. doi:10.1007/978-3-030-72013-1_12.

66 Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979. doi:10.1016/0022-0000(79)90042-4.

67 Grigorii Samuilovich Tseitin. On the complexity of derivation in propositional calculus. *Studies in Mathematics and Mathematical Logic*, 2:115–125, 1968.

68 Allen Van Gelder and Yumi K. Tsuji. Satisfiability testing with more reasoning and less guessing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 559–586. DIMACS/AMS, 1993.

69 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.

70 He-Teng Zhang, Jie-Hong R. Jiang, Luca G. Amarù, Alan Mishchenko, and Robert K. Brayton. Deep integration of circuit simulator and SAT solver. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 877–882. IEEE, 2021. doi:10.1109/DAC18074.2021.9586331.

71 He-Teng Zhang, Jie-Hong R. Jiang, and Alan Mishchenko. A circuit-based SAT solver for logic synthesis. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*, pages 1–6. IEEE, 2021. doi:10.1109/ICCAD51958.2021.9643505.

72 He-Teng Zhang, Jie-Hong R. Jiang, Alan Mishchenko, and Luca G. Amarù. Improved large-scale SAT sweeping. In *Proc. 31st International Workshop on Logic & Synthesis*, 2022.

MaxSAT Resolution with Inclusion Redundancy

Ilario Bonacina
Universitat Politècnica de Catalunya, Barcelona, Spain

Maria Luisa Bonet
Universitat Politècnica de Catalunya, Barcelona, Spain

Massimo Lauria
Sapienza Università di Roma, Italy

Abstract

Popular redundancy rules for SAT are not necessarily sound for MaxSAT. The works of [Bonacina-Bonet-Buss-Lauria'24] and [Ihalainen-Berg-Järvisalo'22] proposed ways to adapt them, but required specific encodings and more sophisticated checks during proof verification. Here, we propose a different way to adapt redundancy rules from SAT to MaxSAT. Our rules do not require specific encodings, their correctness is simpler to check, but they are slightly less expressive. However, the proposed redundancy rules, when added to MaxSAT-Resolution, are already strong enough to capture Branch-and-bound algorithms, enable short proofs of the optimal cost of notable principles (e.g., the Pigeonhole Principle and the Parity Principle), and allow to break simple symmetries (e.g., XOR-ification does not make formulas harder).

2012 ACM Subject Classification Theory of computation → Proof complexity; Theory of computation → Complexity theory and logic

Keywords and phrases MaxSAT, Redundancy, MaxSAT resolution, Branch-and-bound, Pigeonhole principle, Parity Principle

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.7

Funding *Ilario Bonacina*: The author was supported by grant PID2022-138506NB-C22 (PROOFS BEYOND) funded by AEI.
Maria Luisa Bonet: The author was supported by grant PID2022-138506NB-C22 (PROOFS BEYOND) funded by AEI.
Massimo Lauria: The author has been supported by the project PRIN 2022 "Logical Methods in Combinatorics" N. 2022BXH4R5 of the Italian Ministry of University and Research (MIUR).

1 Introduction

MaxSAT is the problem of finding an assignment that minimizes the number of falsified clauses in a given CNF formula. Several variants of MaxSAT exist that, for example, allow to give different weights to clauses, or enforce some clauses to be hard requirements for the solution. While all state-of-the-art SAT-solvers are more or less based on the same theoretical approach, there is more variety among state-of-the-art MaxSAT solvers, e.g., core-guided, minimum-hitting-set, branch-and-bound, and MaxSAT Resolution [27, 4]. Here we focus mostly on MaxSAT Resolution and we make some observations about branch-and-bound (in Section 5). MaxSAT Resolution was first defined in [15] and proved complete for MaxSAT in [12]. Although MaxSAT is a much harder problem than SAT, in some cases MaxSAT solvers can be adapted to be more efficient than CDCL SAT-solvers on hard problems, for instance dual-rail MaxSAT Resolution [10] has short proofs of the Pigeonhole Principle.

We propose new proof systems for MaxSAT by incorporating *redundancy rules* into MaxSAT resolution. Redundancy rules were introduced in SAT solving to allow the introduction of clauses that preserve satisfiability even though they are not logical consequences.

In other words, redundant clauses formalize the notion of reasoning "*without loss of generality*" [29] by restricting the space of solutions without killing it entirely. The first type of redundancy rules considered were Blocked Clauses (BC) [26] and Resolution Asymmetric Tautologies (RAT) [23, 18]. BC, RAT and stronger redundancy rules are extensively used in proof logging of pre-processing and in-processing of state-of-the-art SAT-solvers, and hence are extensively studied in the literature, for instance in [16, 17, 19, 21, 24, 30]. Particularly relevant to this article is the work of Buss and Thapen [13]. Redundancy rules strengthen significantly the Resolution proof system, for instance allowing to prove efficiently the Pigeonhole Principle [21].

Redundancy rules for MaxSAT can only add clauses that do not increase the minimum number of falsified clauses, and the usual redundancy checks used in SAT do not provide any guarantees on that. Recently, some papers have proposed ways to integrate and study redundancy rules in MaxSAT. The work of Ihalainen, Berg and Järvisalo [22], building on [6], studies the advantage of redundancy to preprocess MaxSAT instances. Proof system veriPB [7] includes redundancy rules, among many others. Being rooted in cutting planes, veriPB is particularly apt at certifying optimality, and it can log the reasoning of MaxSAT solver strategies that are way out of reach of MaxSAT resolution [5]. In contrast to these works, the explicit goal of the of Bonacina, Buss, Bonet, Lauria [8] is to study the proof complexity of redundancy rules for MaxSAT in a similar vein of what [13] does for SAT, something that is beyond the scope of those other works, more focused on proof logging actual solvers. To witness that a clause is redundant for SAT it is sufficient to provide an appropriate variable substitution, and even that is not necessary for the weakest forms of redundancy like BC. For MaxSAT, a proof must witness that the redundant clause preserves optimality. In [22] the authors do not provide a polynomially checkable methods for their most powerful rule, but they do for the simplest ones. In [7] they leverage the underlying language cutting planes, usually strong enough to prove the redundancy of a clause explicitly. The work in [8] gives a simple condition that is easy to check and allow to reuse all redundancy rule from [13] in the MaxSAT setting.

The scope of this paper is to understand the power of redundancy rules in the context of MaxSAT and MaxSAT resolution. We analyze forms of redundancy that are simple, and yet add non-trivial power to MaxSAT resolution. In particular, we adapt the rules from [13], namely Literal Propagation Redundancy (LPR), Subset Propagation Redundancy (SPR), Propagation Redundancy (PR), or Substitution Redundancy (SR), to the context of MaxSAT. For simplicity of presentation, we develop our redundancy rules in a setting where all clauses are soft. Everything generalizes easily to a setting with soft/hard clauses. There, the hard clauses would be manipulated by standard resolution, while the soft clauses would be subjected to MaxSAT resolution and our new redundancy rules. That setting would make the arguments more cumbersome, without providing any further technical insight.

We stress again that this paper does not provide strong proof systems for MaxSAT. Even MaxSAT-Resolution +iSR, which the stronger proof system presented here, is easily simulated by veriPB, a very strong proof system that captures most proof techniques in SAT/CP solving and optimization. Our goal is quite the opposite: we add redundancy in the least intrusive way possible to a simple proof system for MaxSAT, and we investigate its strength. In this setting lower bounds could be reachable, at least for weak redundancy rules as iSPR. On the contrary lower bounds for veriPB are way beyond the reach of current techniques.

Moreover, regarding proof logging applications, the framework we propose captures simple branch-and-bound algorithms, it is plausible it could capture *some* branch-and-bound usage from MaxCDCL [28], and can also witness some preprocessing techniques. On the other hand, it is unlikely that MaxSAT-Resolution +iSR could capture core-guided MaxSAT. Nevertheless, a small advantage over veriPB is that DRAT/PR verifiers can be adapted to work with our framework with minimal effort.

The redundancy rules in [22, 8, 7] are all variants of rules LPR, SPR, PR, SR as given in [13]. All these variants rely on a specific, although common, *blocking variables* encoding of MaxSAT instances. Furthermore, every time a redundant clause is added to the proof, the verifier must check that cost is preserved.

In this work we follow a different approach. Essentially, we show that BC and suitable variants of LPR, SPR, PR and SR rules are already cost-preserving, thus no additional check is necessary. Moreover, our approach works even without the blocking variable encoding. To give a concrete idea, when studying satisfiability we say that a clause C is redundant w.r.t. a set of clauses Γ if there exists a partial assignment α that satisfies C and such that $\Gamma|_{\overline{C}} \models \Gamma|_\alpha$; that is, the set of clauses $\Gamma|_{\overline{C}}$, i.e., the clauses of Γ restricted by the assignment which is the negation of C, logically imply all the clauses in $\Gamma|_\alpha$. In other words, if we can find a solution of Γ assuming C is false, there is also a solution of Γ assuming the partial assignment α. Unfortunately the relation $\models$ is not polynomially checkable therefore the relation is substituted with unit-propagation $\vdash_1$, a simpler form of logical implication which is efficiently checkable. This notion is fine for satisfiability, but unfortunately the $\vdash_1$ relation is not cost-preserving: the central idea in this paper is to consider $\subseteq$ instead of $\vdash_1$ in the redundancy condition.

In the context of MaxSAT, we say that a clause C is redundant w.r.t. a *multiset* of clauses Γ, when $\Gamma|_{\overline{C}} \supseteq \Gamma|_\alpha$ for some α that satisfies C. Recall that inclusion between multiset takes multiple occurrences in account. We define rules iLPR, iSPR, iPR and iSR that are the "inclusion" versions of redundancy rules LPR, SPR, PR, and SR (Definition 3.2). There is no inclusion version of BC since it is already defined using inclusion. This change makes the rules immediately cost-preserving, thus avoids the extra check on the cost and the blocking variables encoding, needed in [22, 8, 7]. The relation $\supseteq$ is weaker than $\vdash_1$, therefore, in principle, our rules are formally less expressive. This does not seem to be a limitation: in the context of SAT, all the upper bounds for hard tautologies showed in [13], while described using the SPR rule, do not use unit-propagation but only inclusion, even if the SPR rule would have allowed it.

Adding redundancy rules to the resolution proof system makes it stronger. We add our new rules to the MaxSAT-Resolution proof system (Definition 3.4) and we see that indeed it becomes more powerful: we see that some hard contradictions for MaxSAT-Resolution become easy (the Pigeonhole Principle and the Parity Principle, Theorem 4.3 and Theorem 4.4 resp.), and we show that we can "undo" the effect of xorifications (Theorem 4.5). Both the Pigeonhole Principle and the Parity Principle are exponentially hard for MaxSAT-Resolution, since a MaxSAT-Resolution proof of cost at least 1 is also, syntactically, a Resolution refutation, and the principles above are exponentially hard for Resolution [14].

One goal of proof systems for MaxSAT is to capture the reasoning of MaxSAT solvers and it turns out that the iPR rule is quite apt at logging *Branch-and-Bound* (BnB) approaches to optimization. In its simplest form, the BnB approach explores the possible assignments for a CNF formula in a tree-like fashion, in order to find one that satisfies the largest number of clauses, cutting the branches that are bound to give solutions worse than the best one discovered so far. This works well in many scenarios, but historically has not performed well on MaxSAT industrial instances. Recently, though, a better integration with CDCL has made BnB competitive again [28]. As a proof of concept we show how the basic BnB approach can be simulated in MaxSAT-Resolution+iPR.

Structure of the Paper

Section 2 contains notation and preliminaries. In Section 3 we introduce the redundancy rules for MaxSAT and proof systems for MaxSAT based on them. Section 4 showcases the strength of the system. We give short refutation of the Pigeonhole Principle, the Parity Principle, and show how to undo xor-ifications. In Section 5 we show how to simulate in the system Branch-and-bound algorithms. Finally, Section 6 contains some concluding remarks and open problems.

2 Preliminaries

Basic Notation. For $n \in \mathbb{N}$, let $[n] = \{1, \dots, n\}$. We denote with capital Roman or Greek letters sets and multisets. The *size* of a multiset S is $|S|$, the number of elements in S (counted with multiplicity). Given two multisets S, T, S is *included* in T ($S \subseteq T$) if each element of S appears in T with multiplicity at least the multiplicity it had in S.

A *Boolean variable* x is a variable that takes values $\top$ (true) or $\bot$ (false). A *literal* is a Boolean variable x or its negation $\overline{x}$. We consider fixed a finite set X of Boolean variables and let $\overline{X} = \{\, \overline{x} \mid x \in X \,\}$. A *clause* is a finite disjunction of literals. A clause D is a *weakening* of a clause C if $D = C \vee C'$ for some clause C'. The empty disjunction is $\bot$.

A map $\sigma \colon X \cup \overline{X} \cup \{\top, \bot\} \to X \cup \overline{X} \cup \{\top, \bot\}$ is a *substitution* if σ is the identity on $\{\top, \bot\}$, and for each $\overline{x} \in \overline{X}$, $\sigma(\overline{x}) = \overline{\sigma(x)}$, where $\overline{\top} = \bot$, $\overline{\bot} = \top$, and for each Boolean variable x, $\overline{\overline{x}} = x$. The composition $\sigma \circ \tau$ of two substitutions σ, τ is also a substitution, where $\sigma \circ \tau(v) = \sigma(\tau(v))$ for every $v \in X \cup \overline{X} \cup \{\top, \bot\}$.

A substitution σ is an *assignment* if for every $\ell \in X \cup \overline{X}$, either $\sigma(\ell) = \ell$ or $\sigma(\ell) \in \{\top, \bot\}$. An assignment τ *extends* an assignment σ if $\sigma^{-1}(\{\top, \bot\}) \subseteq \tau^{-1}(\{\top, \bot\})$. An assignment σ is *total* if for every $\ell \in X \cup \overline{X}$, $\sigma(\ell) \in \{\top, \bot\}$. The *domain* of an assignment σ is $\mathrm{dom}(\sigma) = \sigma^{-1}(\{\top, \bot\})$.

Given a clause $\bigvee_{x \in L} \ell$ and a substitution σ, the *restriction* of C under σ is $C|_\sigma = \bigvee_{x \in L} \sigma(\ell)$, simplified using the usual logic rules ($\top \vee \ell = \top$, $\ell \vee \ell = \ell$, etc.). A substitution σ *satisfies* a clause C if $C|_\sigma$ is a *tautology*, i.e., $C|_\sigma$ is a weakening either of $\top$ or $x \vee \overline{x}$ for some variable x. Given a clause $C = \bigvee_{\ell \in L} \ell$, we denote with $\overline{C}$ the assignment given by $\sigma(\ell) = \bot$ and $\sigma(\overline{\ell}) = \top$ if $\ell \in L$ and the identity otherwise. In particular, $C|_{\overline{C}} = \bot$.

Given a set of clauses Γ and a substitution σ, the *restriction* of Γ under σ is the multiset $\Gamma|_\sigma = \{C|_\sigma : C \in \Gamma \text{ and } \sigma \not\models C\}$. A substitution σ *satisfies* a multiset of clauses Γ ($\sigma \models \Gamma$) if σ satisfies all the clauses in Γ, i.e., $\Gamma|_\sigma = \emptyset$. A multiset of clauses Γ *entails* a multiset of clauses Δ if for every substitution σ s.t. $\sigma \models \Gamma$ it holds $\sigma \models \Delta$.

▶ Fact 1. If S, T are multisets of clauses with $S \subseteq T$ and σ is a substitution then $S|_\sigma \subseteq T|_\sigma$.

▶ Fact 2 ([13, Lemma 1.1]). If S is a multiset of clauses and σ, τ are substitutions, then $(S|_\sigma)|_\tau = S|_{\tau \circ \sigma}$.

MaxSAT and MaxSAT-Resolution. Given a multiset of clauses S, MaxSAT asks to find the maximum number of clauses in S that can be simultaneously satisfied. Equivalently, find a total assignment mapping to $\bot$ the smallest number of clauses in S possible. The *cost* of a multiset of clauses S is $\mathrm{cost}(S)$, the minimum of size of $S|_\alpha$ over all possible total assignments α. Notice that for a total assignment α, $S|_\alpha$ is a multiset of the form $\{\bot, \dots, \bot\}$. The goal of proof systems for MaxSAT is to show lower bounds on the cost of MaxSAT instances. One of such systems is MaxSAT-Resolution which was introduced in [15] and proved complete in [12]. We present the system using the rules of [2], first used in the context of MaxSAT in [11].

▶ **Definition 2.1** (MaxSAT-Resolution). *A sequence of multisets $(S_i)_{i\in[m]}$ is a* derivation *of S_m from S_1 in* MaxSAT-Resolution *if for each $i \in [m]$ either*

(i) $S_{i+1} = (S_i \setminus \{C \vee x,\ C \vee \overline{x}\}) \cup \{C\}$ *where* $\{C \vee x,\ C \vee \overline{x}\} \subseteq S_i$ *(SYMM. CUT); or*

(ii) $S_{i+1} = (S_i \setminus \{C\}) \cup \{C \vee x,\ C \vee \overline{x}\}$ *where* $C \in S_i$ *(SPLIT).*

The size *of the derivation is* $\sum_{i\in[m]}|S_i|$*.*

It is known that MaxSAT-Resolution is *sound* and *complete* for MaxSAT:

Soundness Whenever $(S_i)_{i\in[m]}$ is a MaxSAT-Resolution derivation, $\text{cost}(S_1) = \text{cost}(S_m)$. Therefore, if $\perp$ appears in S_m with multiplicity k, then trivially $\text{cost}(S_m) \geq k$ and we say that the MaxSAT-Resolution derivation *certifies* $\text{cost}(S_1) \geq k$, since $\text{cost}(S_1) = \text{cost}(S_m)$.

Completeness Whenever $\text{cost}(S_1) = k$ there is a MaxSAT-Resolution derivation $(S_i)_{i\in[m]}$ where $\perp$ occurs k times in S_m, and all the remaining clauses in S_m are satisfiable [12, Theorem 10].

3 Cost-preserving rules

In the context of SAT, a clause C is *redundant* w.r.t. a set of clauses F if adding C to F does not affect its satisfiability or unsatisfiability, i.e., $F \cup \{C\}$ is satisfiable if and only if F is satisfiable [26]. In particular if $F \models C$, then C is redundant.

In the context of MaxSAT the intuition is similar, a clause C is *redundant* w.r.t. a multiset of clauses S if adding C to S does not affect the cost. In this context, even when $F \models C$, it is not clear that C is redundant w.r.t. S. The notion of redundancy for MaxSAT was introduced in [22] in a slightly different setting than ours.

▶ **Definition 3.1** (redundant clause). *A clause C is* redundant *w.r.t. a multiset of clauses S if* $\text{cost}(S) = \text{cost}(S \cup \{C\})$*.*

Notice that C is redundant w.r.t. S if and only if there exists a total assignment β with $|S|_\beta| = \text{cost}(S)$ and $\beta \models C$. In particular, if C is redundant w.r.t. S, then the clause C could be added to S with arbitrary multiplicity without changing the cost, i.e., $\text{cost}(S) = \text{cost}(S \cup \{C\}) = \text{cost}(S \cup \{C, \dots, C\})$.

Unless $\mathsf{P} = \mathsf{NP}$, it is not polynomially checkable whether a clause C is redundant w.r.t. a multiset of clauses S, therefore, as in the context of SAT, we consider polynomially verifiable notions of redundancy, i.e., ways of adding redundant clauses (as per Definition 3.1) while certifying efficiently their redundancy. In [8] the authors described a systematic way of porting to MaxSAT the notions of efficiently certifiable redundancy already studied in the literature of SAT [20], for example the systems SR/PR/SPR/LPR/BC . This relied on (1) a particular form of the MaxSAT instance considered, and (2) an additional condition to enforce the correctness for MaxSAT.

By considering limited versions of the rules SR/PR/SPR/LPR/BC, we show a conceptually simpler way of adding redundancy rules to MaxSAT. This approach is alternative to the one in [8, 22].

▶ **Definition 3.2** (Inclusion Substitution Redundant, iSR). *A clause C is* Inclusion Substitution Redundant (iSR) *w.r.t. a multiset of clauses S if exists a substitution σ s.t.*

$$(S \cup \{C\})|_\sigma \subseteq S|_{\overline{C}}\ .$$

If the substitution σ has some additional structure, we also have the following redundancy rules, listed in decreasing order of generality:

Inclusion Propagation Redundant (iPR) if σ is an assignment.

Inclusion Subset Propagation Redundant (iSPR) if σ is an assignment with the same domain of $\overline{C}$. Hence σ differs from $\overline{C}$ in the value given to some variables.

Inclusion Literal Propagation Redundant (iLPR) if σ is an assignment with the same domain of $\overline{C}$ and differs from $\overline{C}$ in the value given to exactly one variable.

Blocked Clause (BC) if σ is an assignment with the same domain of $\overline{C}$ and differs from $\overline{C}$ in the value given to exactly one variable x, and moreover for every clause in $D \in S$ containing the variable x, $\sigma \models D$.[1]

Notice that while BC and iLPR might look very similar, they are distinct concepts. For instance, the clause $C = x$ is iLPR w.r.t. $S = \{y, y \vee x, y \vee \overline{x}\}$ but it is not BC w.r.t. the same set. Another redundancy rule is SBC, a generalization of BC defined in [25] which we do not address explicity. As it happens for BC and iLPR, SBC is a valid redundancy rule for MaxSAT and a proper special case of iSPR. For instance, the clause $C = \{x \vee z\}$ is iSPR w.r.t. $F = \{y, y \vee x, y \vee \overline{x}, y \vee x \vee \overline{z}, y \vee \overline{x} \vee z\}$, but it is not SBC (nor iLPR).

▶ **Lemma 3.3.** *If a clause C is* iSR *w.r.t a multiset of clauses S, then C is redundant w.r.t. S, i.e.,* $\text{cost}(S) = \text{cost}(S \cup \{C\})$.

Proof. Clearly $\text{cost}(S) \leq \text{cost}(S \cup \{C\})$. To prove the other inequality, let $k = \text{cost}(S)$ and let β be a total assignment such that $|S|_\beta| = k$. If $\beta \models C$, then we are done. Suppose then $C|_\beta = \bot$. That is β extends $\overline{C}$. By assumption, there is a substitution σ such that $(S \cup \{C\})|_\sigma \subseteq S|_{\overline{C}}$. Therefore $(S \cup \{C\})|_{\beta \circ \sigma} \subseteq S|_{\beta \circ \overline{C}} = S|_\beta$. Hence $|(S \cup \{C\})|_{\beta \circ \sigma}| \leq |S_\beta| = k$, and $\text{cost}(S \cup \{C\}) \leq k$. ◀

Checking whether a clause is iSR w.r.t. S, given the substitution σ, is doable in polynomial time. Therefore we can extend any proof system for MaxSAT with a rule that introduces iSR clauses. We now consider such extension for MaxSAT-Resolution.

▶ **Definition 3.4** (MaxSAT-Resolution + iSR). *A sequence of multisets $(S_i)_{i \in [m]}$ is a* derivation *of S_m from S_1 in* MaxSAT-Resolution + iSR *if for each $i \in [m]$ either one of the cases* (i), (ii) *of the definition of* MaxSAT-Resolution *occur, or*

(iii) $S_{i+1} = S_i \cup \{C\}$ *where C is* iSR *w.r.t. S_i;*

(iv) $S_{i+1} = S_i \setminus \{C\}$ *where C is* iSR *w.r.t. $S_i \setminus \{C\}$.*

Each occurrence of the rules (iii) and (iv) is accompanied by the corresponding substitution σ witnessing the validity of the rule. The size *of the derivation is $\sum_{i \in [m]} |S_i|$. The definition of* MaxSAT-Resolution+R *for any $R \in \{$*iPR, iSPR, iLPR, BC*$\}$ is analogous.*

We only consider the case where MaxSAT-Resolution + iSR derivation are not allowed to introduce new variables, since introducing new variables makes the systems as strong as Extended Resolution [26]. To be consistent with [13], the rules/systems should be called MaxSAT-Resolution+ iSR$^-$, where the "$-$" is used to indicate that the systems are not allowed to introduce new variables. We ignore that convention to ease notation. The system MaxSAT-Resolution + iSR is *sound* and *complete.*

Soundness Lemma 3.3 and the soundness of MaxSAT-Resolution immediately imply that MaxSAT-Resolution + iSR is also sound, i.e., whenever $(S_i)_{i \in [m]}$ is a MaxSAT-Resolution + iSR derivation, $\text{cost}(S_1) = \text{cost}(S_m)$. Therefore, as in the case of MaxSAT-Resolution, we say that the MaxSAT-Resolution + iSR derivation *certifies* $\text{cost}(S_1) \geq k$ if S_m contains $\bot$ with multiplicity at least k.

Completeness The completeness of MaxSAT-Resolution + iSR is immediate from the completeness of the system MaxSAT-Resolution.

[1] This is not the usual definition of BC but it is equivalent, as shown in [8].

▶ Remark 3.5. The choice of MaxSAT-Resolution in Definition 3.4 is in some sense arbitrary: the rule iSR (i.e., items (iii) and (iv) in Definition 3.4) could be added or easily adapted to any sound proof system for MaxSAT with substitution rules, for instance, the weighted resolution proof system from [11]. In particular, since weighted resolution is equivalent to Sherali-Adams, and restricted weighted resolution is equivalent to Nullstellensatz [11, 9], this means the iSR rule could be also added to those (semi-)algebraic proof systems.

▶ Remark 3.6. Our goal is to adapt rules like SR from the SAT framework to MaxSAT. Nevertheless the iSR rule and its restrictions iPR/iSPR/iLPR/BC still make sense for SAT. If we apply them to sets instead of multisets, they immediately become special cases of the original redundancy rules for SAT. For example iSR in this context is a special case of the SR rule that we spell here for convenience. A clause C is *Substitution Redundant* (SR) w.r.t. a set of clauses S if

$$S|_{\overline{C}} \vdash_1 (S \cup \{C\})|_\sigma \ ,$$

where $\vdash_1$ indicates *unit propagation*, an efficiently checkable form of entailment. To the best of our knowledge, the iSR/iPR/iSPR/iLPR rules are presented here for the first time, both in the context of SAT and MaxSAT. We observe, though, that the upper bounds for pigeonhole principle, bit-pigeonhole principles, clique-coloring, parity, xor-ification, and Tseitin formulas in [13, Section 4] are stated for the rule SPR in the case of SAT, but in fact fulfill the inclusion condition as in iSPR. On the other hand, these results cannot be automatically adapted to MaxSAT-Resolution+iSPR due to the MaxSAT-Resolution rule restrictions.

4 Certifying the cost of some hard tautologies

In this section we exemplify the power of MaxSAT-Resolution + iSR by (1) efficiently certifing the optimum cost of the pigeonhole principle PHP^m_n, (2) efficiently certifying the optimum cost of the parity principle, and (3) reversing the hardness increase due to xor-ification of CNFs. To do so we use few simple, yet useful, lemmas.

▶ **Lemma 4.1.** *Given a clause C and multisets of clauses S and T, if there is a substitution π such that $S|_\pi \subseteq S$ and $C|_\pi \vee C$ is a tautology and for every clause $D \in T$, $D|_\pi \vee C$ is also a tautology, then C is* iSR *w.r.t. $S \cup T$.*

Proof. Let π such that $S|_\pi \subseteq S$ and $C|_\pi \vee C$ is a tautology. Let $\sigma = \overline{C} \circ \pi$. Since $C|_\pi \vee C$ is a tautology we have $\overline{C} \models C|_\pi$ and $\sigma \models C$. Similarly for every clause $D \in T$, $\sigma \models D$. That is

$$(S \cup T \cup \{C\})|_\sigma = S|_\sigma = S|_{\overline{C}\circ\pi} = (S|_\pi)|_{\overline{C}} \subseteq S|_{\overline{C}} \subseteq (S \cup T)|_{\overline{C}} \ . \qquad ◀$$

It is well known that pure literals are blocked clauses. For convenience here we state essentially the same fact for iLPR.

▶ **Lemma 4.2.** *Given a multiset of clauses Γ and a literal ℓ such that $\overline{\ell}$ does not occur in Γ. We can derive Γ' from Γ using the* iLPR *rule where $\Gamma' \subseteq \Gamma$ and Γ' is a multiset not containing any clause with the literal ℓ.*

Proof. It is sufficient to consider the symmetric difference to be a single clause $C \vee \ell$ and show that $C \vee \ell$ is iLPR w.r.t. to Γ. The main claim follows by repeated applications of iLPR rules.

Let $\Gamma = \Gamma_0 \cup \Gamma_\ell$ where Γ_0 are the clauses containing neither ℓ nor $\overline{\ell}$, and Γ_ℓ are the clauses with literal ℓ. We fix $\sigma = \{\overline{C} \wedge \ell = 1\}$, and observe that

$$(\Gamma \cup \{C \vee \ell\})|_\sigma = \Gamma_0|_{\overline{C}} = \Gamma_0|_{\overline{C}\wedge\ell=0} \subseteq \Gamma|_{\overline{C}\wedge\ell=0} \ . \qquad ◀$$

4.1 Pigeonhole principle PHP^m_n

Let $m, n \in \mathbb{N}$ with $m > n$. The propositional encoding of the Pigeonhole Principle PHP^m_n uses Boolean variables $p_{i,j}$ with $i \in [m]$ and $j \in [n]$ with intended meaning that $p_{i,j}$ is true if and only if the pigeon i flies to hole j. For $i < k$ let the *injectivity axiom* $\mathsf{Inj}_{i,k,j}$ be the clause $\overline{p_{i,j}} \vee \overline{p_{k,j}}$, expressing that the two pigeons i, k cannot fly at the same time to hole j; and let the *totality axiom* $\mathsf{Tot}_{i,n}$ be the clause $\bigvee_{j\in[n]} p_{i,j}$, expressing that the pigeon i must fly somewhere among the n holes. The CNF encoding of Pigeonhole Principle is

$$\mathsf{PHP}^m_n = \{\mathsf{Tot}_{i,n} \mid i \in [m]\} \cup \{\mathsf{Inj}_{i,k,j} \mid i, k \in [m],\ j \in [n] \text{ and } i < k\} .$$

An assignment that maps the first n pigeons to the n holes, and leaves the other pigeons unassigned, falsifies $m - n$ totality axioms and no injectivity axioms. Hence $\mathrm{cost}(\mathsf{PHP}^m_n) \leq m - n$, and we can prove that this is optimal in $\mathsf{MaxSAT\text{-}Resolution} + \mathsf{iSR}$.

▶ **Theorem 4.3.** *There is a polynomial size derivation in* $\mathsf{MaxSAT\text{-}Resolution} + \mathsf{iSR}$ *showing that* $\mathrm{cost}(\mathsf{PHP}^m_n) \geq m - n$.

Proof. It is enough to show how to derive PHP^{m-1}_{n-1} from PHP^m_n, since repeating this process n times gives PHP^{m-n}_0. This latter formula contains no other clauses than $m - n$ totality axioms $\mathsf{Tot}_{i,0}$, that are indeed copies of the empty clause $\perp$. This would conclude the proof.

To derive PHP^{m-1}_{n-1} from PHP^m_n we use iSR to enforce one by one all the pigeons of index below m not to fly into hole n. Namely for $1 \leq i \leq m$ we have the intermediate sets

$$\Gamma_i = \big(\big(\mathsf{PHP}^m_n \setminus \{\mathsf{Inj}_{\ell,k,n}\ :\ \ell < i \text{ and } k \neq \ell\}\big) \setminus \{\mathsf{Tot}_{\ell,n}\ :\ \ell < i\}\big) \cup \{\mathsf{Tot}_{\ell,n-1}\ :\ \ell < i\} ,$$

In particular, $\Gamma_1 = \mathsf{PHP}^m_n$ and $\Gamma_m = \mathsf{PHP}^{m-1}_{n-1} \cup \{\mathsf{Tot}_{m,n}\}$. The variable $x_{m,n}$ appears only in $\mathsf{Tot}_{m,n}$ therefore, by Lemma 4.2, this clause can be removed from Γ_m to get PHP^{m-1}_{n-1}.

Suppose now $1 \leq i < m$, we have the database of clauses is Γ_i and we want to obtain Γ_{i+1}.

Step 1. The clause $C = p_{m,n} \vee \overline{p_{i,n}}$ is iSR for Γ_i, and we witness that with permutation π that exchanges pigeons i and m. That is, $\pi(p_{m,j}) = p_{i,j}$, $\pi(p_{i,j}) = p_{m,j}$, $\pi(\overline{p_{m,j}}) = \overline{p_{i,j}}$ and $\pi(\overline{p_{i,j}}) = \overline{p_{m,j}}$ for every $j \in [n]$. On the other variables π is the identity. The permutation π maps Γ_i to itself: totality axioms $\mathsf{Tot}_{m,n}$ and $\mathsf{Tot}_{i,n}$ are both in Γ_{i-1} and get swapped; the injectivity axioms $\mathsf{Inj}_{\ell,k,n}$ in Γ_i all have $\ell, k \geq i$, therefore π maps this set of axioms to itself, the set of remaining axioms is also mapped to itself. Applying Lemma 4.1 with $T = \emptyset$ we get that $p_{m,n} \vee \overline{p_{i,n}}$ is iSR wrt Γ_i. Once we add $p_{m,n} \vee \overline{p_{i,n}}$ to Γ_i, we cut it with the injectivity axiom $\mathsf{Inj}_{i,m,n}$ to get $\overline{p_{i,n}}$. Now the database of clauses is $\Gamma_i \setminus \{\mathsf{Inj}_{i,m,n}\} \cup \{\overline{p_{i,n}}\}$.

Step 2. To cut $\overline{p_{i,n}}$ with $\mathsf{Tot}_{i,n}$, i.e., $\bigvee_{j\in[n]} p_{i,j}$ we need first to split $\overline{p_{i,n}}$ repeatedly getting the database of clauses

$$\big(\Gamma_i \setminus \{\mathsf{Inj}_{i,m,n}\}\big) \cup \{\overline{p_{i,n}} \vee \big(\bigvee_{\ell\in[j-1]} p_{i,\ell}\big) \vee \overline{p_{i,j}}\ :\ j \in [n-1]\} \cup \{\overline{p_{i,n}} \vee p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n-1}\} .$$

Now cut $\overline{p}_{i,n} \vee p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n-1}$ with the totality axiom $\mathsf{Tot}_{i,n}$ to obtain $\mathsf{Tot}_{i,n-1}$ and the database of clauses

$$\Delta = \big(\Gamma_i \setminus \{\mathsf{Inj}_{i,m,n}, \mathsf{Tot}_{i,n}\}\big) \cup \{\mathsf{Tot}_{i,n-1}\} \cup \{\overline{p_{i,n}} \vee \big(\bigvee_{\ell\in[j-1]} p_{i,\ell}\big) \vee \overline{p_{i,j}}\ :\ j \in [n-1]\} .$$

Step 3. The database Δ only contains variable $p_{i,n}$ with negative polarity, hence, Lemma 4.2 allows to remove all the clauses containing $\overline{p_{i,n}}$, and get the database of clauses

$$\big((\Gamma_i \setminus \{\mathsf{Inj}_{i,k,n} \ : \ i < k\}) \setminus \{\mathsf{Tot}_{i,n}\}\big) \cup \{\mathsf{Tot}_{i,n-1}\} = \Gamma_{i+1} \ . \qquad ◀$$

The proof of Theorem 4.3 is a generalization of the argument to prove efficiently the unsatisfiability of PHP_n^{n+1} in [13, Example 1.4], which in turn is based on [20].

4.2 Parity principle

The *Parity Principle* claims that there is a perfect matching between an odd number of elements. The propositional encoding of this principle (Parity_n) is minimally unsatisfiable, and here we show that it has a short proof in MaxSAT-Resolution+ iSR. This is interesting since the formula is hard for Sherali-Adams and Sum-of-Squares proof systems [3, 1].

The set of clauses Parity_n has Boolean variables $x_{\{i,j\}}$ for $i, j \in [n]$ with $i \neq j$, where $x_{\{i,j\}}$ means that elements i and j are matched together. To ease the notation we use $x_{i,j}$ and $x_{j,i}$ as alternative notations for $x_{\{i,j\}}$. For each $i, n' \in [n]$ we define the set of clauses

$$\mathsf{AtLeast}_i^{n'} = \bigvee_{j \in [n'] \setminus \{i\}} x_{i,j} \qquad \mathsf{AtMost}_i^{n'} = \{\overline{x}_{i,j} \vee \overline{x}_{i,j'} \ : \ j, j' \in [n'],\ i, j, j' \text{ all distinct}\} \ .$$

Their informal meaning is that the element i matches with at least and at most one distinct element in $[n']$ respectively. The set of clauses Parity_n is then

$$\mathsf{Parity}_n = \{\mathsf{AtMost}_i^n,\ \mathsf{AtLeast}_i^n \ : \ i \in [n]\} \ .$$

For n odd, there is an assignment of the variables satisfying all but one clause, that is Parity_n is minimally unsatisfiable.

▶ **Theorem 4.4.** *For odd n, there is a polynomial size derivation in* MaxSAT-Resolution + iSR *showing that* $\mathrm{cost}(\mathsf{Parity}_n) \geq 1$.

Proof. The strategy of the proof is to start with a clauses of Parity_n and deduce from it the clauses of Parity_{n-2}. Since n is odd at some point we get to Parity_1, which contains $\mathsf{AtLeast}_1^1$ which is the empty disjunction, i.e., $\perp$.

To reduce Parity_n to Parity_{n-2} we enforce the elements $n-1$ and n to match.

Step 1. Derive clauses $C_i = \overline{x_{n,i}} \vee x_{n,(n-1)}$ for every $1 \leq i \leq n-2$ in this order one by one. To derive the clause C_i we use Lemma 4.1 with $S = \mathsf{Parity}_n$ and $T = \{C_j \ : \ j < i\}$. As witnessing substitution we use π, the variable permutation induced by swapping indices i and $n-1$. By symmetry $\mathsf{Parity}_n|_\pi = \mathsf{Parity}_n$. Then, observe that the clause $C_i|_\pi$ and all clauses $C_j|_\pi$ for $C_j \in T$ contain the literal $x_{n,i}$, while C_i contains $\overline{x_{n,i}}$. Therefore $C_i|_\pi \vee C_i$ and all $C_j|_\pi \vee C_i$ are tautologies and Lemma 4.1 applies.

Step 2. Derive clauses $D_i = \overline{x_{(n-1),i}} \vee x_{(n-1),n}$ for every $1 \leq i \leq n-2$ in this order, using almost the same strategy of Step 1. To derive the clause D_i we again use Lemma 4.1 with substitution π induced by the variable permutation induced by swapping indices i and n. Everything works as in the previous paragraph, $S = \mathsf{Parity}_n$ and

$$T = \{D_j \ : \ j < i\} \cup \{C_k \ : \ k \in [n-2]\} \ .$$

Clause $D_i|_\pi$ contains the literal $x_{(n-1),i}$ and the same happens for all $D_j|_\pi$ with $j < i$, and for all $C_k|_\pi$ with $k \in [n-2]$. Hence, as in Step 1, all $D_j|_\pi \vee D_i$ with $j \leq i$ and all $C_k|_\pi \vee D_i$ with $k \in [n-2]$ are tautologies, and Lemma 4.1 applies. The current database of clauses is

$$\Gamma = \mathsf{Parity}_n \cup \{C_i \ : \ i \in [n-2]\} \cup \{D_j \ : \ j \in [n-2]\} \ .$$

Step 3. For all $i \in [n-2]$ we do a symmetric cut between the clause $C_i = \overline{x_{n,i}} \vee x_{n,(n-1)}$, introduced in Step 2, and the Parity_n clause $\overline{x_{n,i}} \vee \overline{x_{n,(n-1)}}$, to obtain the set of unit clauses

$$\{\overline{x_{n,i}} \ : \ i \in [n-2]\} \ ,$$

consuming all the clauses of the form C_i and all the clauses $\overline{x_{n,i}} \vee \overline{x_{n,(n-1)}}$ with $i \in [n-2]$. Similarly, for all $j \in [n-2]$ we do a symmetric cut between the clause $D_j = \overline{x_{(n-1),j}} \vee x_{(n-1),n}$, introduced in Step 1, and the Parity_n clause $\overline{x_{(n-1),j}} \vee \overline{x_{(n-1),n}}$, to obtain the set of unit clauses

$$\{\overline{x_{(n-1),j}} \ : \ j \in [n-2]\} \ ,$$

consuming from Γ all the clauses of the form D_j and all the clauses $\overline{x_{(n-1),j}} \vee \overline{x_{(n-1),n}}$ with $j \in [n-2]$. As a result the current database of clauses is

$$\Gamma' = (\mathsf{Parity}_n \setminus \{\overline{x_{(n-1),i}} \vee \overline{x_{(n-1),n}},\ \overline{x_{n,i}} \vee \overline{x_{n,(n-1)}} \ : \ i \in [n-2]\}) \cup \{\overline{x_{i,(n-1)}},\ \overline{x_{i,n}} \ : \ i \in [n-2]\} \ .$$

Step 4. In Γ' literal $\overline{x_{n,n-1}}$ does not occur, so we can use Lemma 4.2 to remove both $\mathsf{AtLeast}^n_{n-1}$ and $\mathsf{AtLeast}^n_n$ from Γ'. The clause database becomes

$$\Gamma'' = \{\mathsf{AtMost}^{n-2}_i, \mathsf{AtLeast}^n_i \ : \ i \in [n-2]\} \cup \{\overline{x_{i,(n-1)}},\ \overline{x_{i,n}} \ : \ i \in [n-2]\} \ .$$

Step 5. To conclude the derivation of Parity_{n-2} we need to shorten the all the clauses $\mathsf{AtLeast}^n_i$ into $\mathsf{AtLeast}^{n-2}_i$ for each $i \in [n-2]$. We show how to derive $\mathsf{AtLeast}^{n-1}_i$ from unit $\overline{x_{i,n}}$ and $\mathsf{AtLeast}^n_i$. The same procedure then works to get $\mathsf{AtLeast}^{n-2}_i$ from unit $\overline{x_{i,n-1}}$ and $\mathsf{AtLeast}^{n-1}_i$.

First split $\overline{x_{i,n}}$ into $\overline{x_{i,1}} \vee \overline{x_{i,n}}$ and $x_{i,1} \vee \overline{x_{i,n}}$, then the latter clause into $x_{i,1} \vee \overline{x_{i,2}} \vee \overline{x_{i,n}}$ and $x_{i,1} \vee x_{i,2} \vee \overline{x_{i,n}}$, and so on up to get $\bigvee_{j \in [n-1] \setminus \{i\}} x_{i,j} \vee \overline{x_{i,n}}$. We do symmetric cut between this last clause and $\mathsf{AtLeast}^n_i$ to get $\mathsf{AtLeast}^{n-1}_i$. Notice that all intermediate clauses from the splits left in the clause database contain the literal $\overline{x_{i,n}}$.

Repeating this procedure using the unit clause $\overline{x_{i,n-1}}$ and $\mathsf{AtLeast}^{n-1}_i$ gives $\mathsf{AtLeast}^{n-2}_i$ and several intermediate clauses containing literal $\overline{x_{i,n-1}}$.

We do this for every $i \in [n-2]$, so that in the clause database we have Parity_{n-2} plus clauses containing literals of the form either $\overline{x_{i,n-1}}$ or $\overline{x_{i,n}}$. We can remove all such clauses using Lemma 4.2 because the opposites of these literals do not occur in Parity_{n-2}. This concludes the derivation of Parity_{n-2}. ◀

Therefore, a consequence of Theorem 4.4 is that neither Sherali-Adams nor Sum-of-Squares as proof systems for MaxSAT can simulate $\mathsf{MaxSAT\text{-}Resolution} + \mathsf{iSR}$.

4.3 XOR-ification

We show that MaxSAT-Resolution+iSPR can "undo" the effect of common techniques used to make hard instances of propositional tautologies. For concreteness we do it for xor-ifications. This is analogous to the case of SAT, where the SPR rule can be used to "undo" the effects of xor-ifications [13, Section 4.6].

Given a multiset of clauses F, the mth *xor-ification* of a variable x is the set of clauses where the variable x is substituted by the XOR of m new variables $x_1 \oplus \cdots \oplus x_m$ and the resulting formula is expanded again as a CNF formula. The mth xor-ification of F (denoted $F[\oplus^m]$) is the procedure above applied to all the variables of F. Notice that $\mathrm{cost}(F) = \mathrm{cost}(F[\oplus^m])$.

▶ **Theorem 4.5.** *Let F be a multiset of clauses with a* MaxSAT-Resolution *derivation showing that* $\mathrm{cost}(F) \geq k$ *in size s, then there is a* MaxSAT-Resolution+iSPR *derivation showing that* $\mathrm{cost}(F[\oplus^m]) \geq k$ *of size polynomial in s and the number of clauses of $F[\oplus^m]$.*

Proof. The idea is to remove all the symmetries among the xor-ified variables at the beginning one by one, and then do the MaxSAT-Resolution derivation. We show how to "undo" the xor-ification of a variable x xor-ified into $x_1 \oplus x_2 \oplus \cdots \oplus x_m$. Without loss of generality we assume m to be even, therefore for each clause in F of the form $C \vee x$, $F[\oplus^m]$ contains the clause $C \vee x_1 \vee x_2 \vee \cdots \vee x_m$; similarly, for each clause in F of the form $C \vee \overline{x}$, $F[\oplus^m]$ contains the clause $C \vee \overline{x_1} \vee x_2 \vee \cdots \vee x_m$. Let ℓ be the number of occurrences of the variable x in F.

Step 1. Let $\Gamma_i = F[\oplus^m] \cup \{\underbrace{\overline{x_j} \ldots \overline{x_j}}_{\ell \text{ copies}} : 2 \leq j \leq i\}$, so that $\Gamma_1 = F[\oplus^m]$. For $i \geq 2$, we see how to derive Γ_i from Γ_{i-1}.

The clauses $x_1 \vee \overline{x_i}$ is iSPR w.r.t. Γ_{i-1} and $\overline{x_1} \vee \overline{x_i}$ is iSPR w.r.t. $\Gamma_{i-1} \cup \{x_1 \vee \overline{x_i}\}$. For the first application of the iSPR rule we set $\sigma = \{x_1 := \top, x_i := \bot\}$. To check that it is a valid application notice that for any clause $C \vee (\overline{x_1} \vee x_i \vee \ldots)$ that gets restricted but not satisfied in $\Gamma_{i-1}|_\sigma$, there is another clause $C \vee (x_1 \vee \overline{x_i} \vee \ldots)$ that gets restricted in the same way in $\Gamma_{i-1}|_{\{x_1:=\bot, x_i:=\top\}}$. For the second application $\sigma = \{x_1 := \bot, x_i := \bot\}$ and we apply a simular reasoning. In both cases we add $x_1 \vee \overline{x_i}$ and $\overline{x_1} \vee \overline{x_i}$ to Γ_{i-1} with multiplicity ℓ. Afterward, by symmetric cut we obtain ℓ copies of $\overline{x_i}$, and this gives us Γ_i. We keep going until we get to Γ_m.

Step 2. Now in the clause database we have ℓ copies each of the sequence of unit clauses $\overline{x}_2, \ldots, \overline{x}_m$. A positive occurrence of original variable x in a clause $C \vee x \in F$ induces a clause $C \vee x_1 \vee x_2 \vee \cdots \vee x_m \in F[\oplus^m]$. To resolve that with $\overline{x}_2, \ldots, \overline{x}_m$ we first apply splits to the units, and eventually we can apply a series of symmetric cuts and obtain $C \vee x_1$. In the same way, for $C \vee \overline{x} \in F$ we work on the corresponding clause $C \vee \overline{x_1} \vee x_2 \vee \cdots \vee x_m$ to get clause $C \vee \overline{x_1}$. In the end the clause database contains a copy of F up to variable renaming. ◀

5 Simulating branch-and-bound with MaxSAT redundancy rules

Given a set of clauses F, the basic *branch-and-bound* (BnB) procedure explores the space of all possible assignments for F in a depth-first way. At every node, the BnB procedure has computed an upper bound UB and a lower bound LB: the UB is the cost of the best solution found so far, while the LB is the number of falsified clauses in the current branch. At the beginning of the procedure UB is the number of clauses in F and the LB is 0. At each node the procedure compares LB and UB at that node: if $LB \geq UB$ the algorithm prunes the branch, i.e., it does not continue to explore the subtree and backtracks to a previous node, since we are exploring an assignment that we already discovered it is not optimal. If $LB < UB$ the algorithm instantiates one more variable and continues the exploration. The solution is the value of UB after exploring the whole search tree.

Here we simulate the basic BnB approach via MaxSAT-Resolution+ iPR. Let S be a multiset of clauses, let T be the BnB decision tree for S and let t be the number of leaves of T. We identify the leaves of T with the partial assignments $\beta_1, \beta_2, \ldots, \beta_t$ that label the branches of the tree, enumerated according to the visit order. Each leaf has an associated cost k_i, which is the LB at node i, the number of clauses of S falsified by β_i. Each leaf β_i is of one of two types:

- *Pruning:* when $k_i \geq k_j$ for some $1 \leq j < i$; k_j corresponds to the UB;
- *Improvement:* when $k_i < k_j$ for all $1 \leq j < i$ and each clause in S is either satisfied or falsified by β_i, because β_i is a leaf. In this case k_i will be the new UB.

▶ **Theorem 5.1.** *Consider a BnB procedure for MaxSAT on clauses S and let T be the BnB decision tree associated showing* $\mathrm{cost}(S) = k$. *Then there is a* MaxSAT-Resolution+ iPR *proof of length $O(k \cdot |T|)$ that* $\mathrm{cost}(S) \geq k$.

Proof. Let t be the number of leaves of T, and $m = |S|$. *First phase:* the proof simulates the BnB by considering the leaves $\beta_1, \ldots, \beta_t$ one by one. The derivation maintains a multiset of clauses that forbid all leaves seen so far, except for the leaf with the current best value. If the next leaf is pruned, then the proof adds a clause to forbid it. If the next leaf is an improvement, then the proof forbids the leaf corresponding to the previous best value. We will use several times the following fact.

▶ Fact 3. For any $i \neq j$, assignment β_j satisfies clause $\overline{\beta}_i$, since β_i and β_j must disagree on the value of some variable.

For $1 \leq i \leq t$, we show how to get the clause database $\Gamma_i = S \cup \{\overline{\beta}_1, \ldots, \overline{\beta}_i\} \setminus \{\overline{\alpha}_i\}$ where α_i is the assignment corresponding to the leaf of minimum cost among $\{\beta_1, \ldots, \beta_i\}$. Leaf β_1 is trivially an improvement, hence $\Gamma_1 = S$. To derive Γ_{i+i} from Γ_i we deal with the two types of leaves separately.

If β_{i+1} was pruned, we forbid it by adding $\overline{\beta}_{i+1}$ to the database Γ_i using the iPR rule with current best assignment α_i as witness. We need to check that $(\Gamma_i \cup \{\overline{\beta}_{i+1}\})|_{\alpha_i} \subseteq \Gamma_i|_{\beta_{i+1}}$. By the previous Fact, all clauses in $\Gamma_i \setminus S = \{\overline{\beta}_1, \ldots, \overline{\beta}_i\} \setminus \{\overline{\alpha}_i\}$ are satisfied both by α_i and by β_{i+1}, and furthermore $\overline{\beta}_{i+1}|_{\alpha_i} = \top$. The check reduces to verifying that $S|_{\alpha_i} \subseteq S|_{\beta_{i+1}}$. The right hand side $S|_{\beta_{i+1}}$ contains k_{i+1} copies of $\perp$ by definition (together possibly with other clauses), and $S|_{\alpha_i}$ contains only the clause $\perp$ with multiplicity at most k_{i+1}.

If leaf β_{i+1} corresponds to an improvement, we forbid α_i by adding $\overline{\alpha}_i$ to the database via iPR using β_{i+1} as witness, that is $(\Gamma_i \cup \{\overline{\alpha}_i\})|_{\beta_{i+1}} \subseteq \Gamma_i|_{\alpha_i}$. The procedure is the same as before, but with the role of α_i and β_{i+1} reversed. In this case $\Gamma_{i+1} = \Gamma_i \cup \{\overline{\alpha}_i\}$.

Second phase: Let α be the assignment corresponding to the optimal leaf, and k be its cost. The clause database now contains $S \cup \{\overline{\beta}_1 \ldots, \overline{\beta}_t\} \setminus \{\overline{\alpha}\}$. Our goal now is to derive k copies of all clauses forbidding all assignments at the leaves of T. From there, we derive k copies of $\perp$ by doing symmetric cuts on the tree branches, and that would conclude the proof.

To derive the missing $k-1$ copies of clauses $\overline{\beta}_i$, for $\beta_i \neq \alpha$, we use the iPR rule with α as witnessing substitution. This is similar to the pruning step, but the right hand side of the inclusion has even more copies of $\perp$. These applications of iPR rule are correct regardless of their order in the proof.

Now we derive k copies of $\overline{\alpha}$: S contains k clauses falsified by α, hence $\overline{\alpha}$ is a weakening of each of them. We do not have a weakening rule in our proof system, but we can simulate it using the split rule. We explain with an example how we do it: let us say that $\overline{\alpha} = x_1 \vee x_2 \vee \cdots \vee x_\ell$ and that $C = x_1 \vee x_2 \vee \cdots \vee x_j \in S$ for some $j \leq \ell$. We use the split rule on C to get $C \vee \overline{x}_{j+1}$ and $C \vee x_{j+1}$, then on the latter to get $C \vee x_{j+1} \vee \overline{x}_{j+2}$ and $C \vee x_{j+1} \vee x_{j+2}$, and so on. In this way we derive a copy of $\overline{\alpha}$ from each of the k falsified clauses, each of them in $O(|\alpha|)$ steps.

In total we do $k(|T|-1)$ applications of iPR rule, and $O(k|\alpha|)$ to derive the clauses corresponding to the optimal leaf. Finally we get the k empty clauses in $O(|T|)$ steps each, resolving bottom to top in the tree using symmetric cuts. ◀

The division of the proof into first and second phase looks artificial, but we used it to highlight how part of the proof can be logged during the BnB procedure. In the second phase we use the optimum k, that is only known at the end of the procedure. As an alternative, for any non-optimal branch $\beta_i \neq \alpha$ we could produce a sufficient number of copies of $\overline{\beta}_i$ when we add it, and use only k of them at the end, since the optimal branch α can only be produced with multiplicity k (as in the previous proof). The sufficient number of copies could be the current UB of the branch, which is always greater or equal than k. This, on the other hand, makes the proof longer, but could be avoided using the language of weighted clauses.

6 Conclusions and open problems

We convert redundancy rules SR/PR/SPR/LPR for SAT into rules iSR/iPR/iSPR/iLPR that are sound for MaxSAT. Adding such rules to MaxSAT-Resolution produces new proof systems. We exemplify their strength with short proofs of the optimal cost of hard tautologies (Section 4), and with a simulation of simple BnB procedures (Section 5). We conclude with a list of open problems.

- Can MaxSAT-Resolution+iSPR prove either $\text{cost}(\mathsf{PHP}^m_n) = m - n$ or $\text{cost}(\mathsf{Parity}_n) = 1$ efficiently? Polynomial-size proofs of the unsatisfiability of PHP^{n+1}_n and Parity_n are known for the system SPR$^-$ [13] but the arguments don't seem to adapt to MaxSAT-Resolution +iSPR.
- Sparse versions of Pigeonhole Principle allow each pigeon to only fly into a small selection of holes, i.e., some variables $p_{i,j}$ are set to $\perp$. In proof systems that are closed under variable restrictions, the sparse version is at least as easy as the standard version. But this closure property does not hold in proof using redundancy rules, thus it is interesting to ask whether sparse versions of pigeonhole principle are easy for MaxSAT-Resolution + iSR or SR$^-$.
- How does MaxSAT-Resolution + iSR compare with MaxSAT-Resolution+cost-SR from [8]?
- We simulate a plain BnB in MaxSAT-Resolution + iPR. This suggests that iPR rule could be instrumental to simulate more sophisticated BnB algorithms. For example algorithms that integrate CDCL reasoning [28].

References

1 Albert Atserias and Tuomas Hakoniemi. Size-degree trade-offs for Sums-of-Squares and Positivstellensatz proofs. In *34th Computational Complexity Conference (CCC)*, volume 137, pages 24:1–24:20, 2019. doi:10.4230/LIPIcs.CCC.2019.24.

2 Albert Atserias and Massimo Lauria. Circular (yet sound) proofs in propositional logic. *ACM Trans. Comput. Log.*, 24(3):20:1–20:26, 2023. Conference version appeared in SAT'19. doi:10.1145/3579997.

3 Per Austrin and Kilian Risse. Perfect matching in random graphs is as hard as Tseitin. *TheoretiCS*, 1, 2022. doi:10.46298/THEORETICS.22.2.

4 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. doi:10.3233/FAIA201008.

5 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, , and Dieter Vandesande. Certified core-guided maxsat solving. In *Proceedings of the 29th International Conference on Automated Deduction (CADE-29), July 2023*, 2023.

6 Jeremias Berg and Matti Järvisalo. Unifying reasoning and care-guided search for maximum satisfiability. In *16th European Conf. on Logics in Artificial Intelligence (JELIA)*, pages 287–303, 2019.

7 Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.*, 77:1539–1589, 2023. doi:10.1613/JAIR.1.14296.

8 Ilario Bonacina, Maria Luisa Bonet, Sam Buss, and Massimo Lauria. Redundancy rules for MaxSAT. *Electron. Colloquium Comput. Complex.*, TR24-045, 2024. URL: https://eccc.weizmann.ac.il/report/2024/045.

9 Ilario Bonacina, Maria Luisa Bonet, and Jordi Levy. Weighted, circular and semi-algebraic proofs. *Journal of Artificial Intelligence Research (JAIR)*, 79:447–482, February 2024. doi:10.1613/jair.1.15075.

10 Maria Luisa Bonet, Sam Buss, Alexey Ignatiev, João Marques-Silva, and Antonio Morgado. MaxSAT resolution with the dual rail encoding. In *32nd Intl. AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

11 Maria Luisa Bonet and Jordi Levy. Equivalence between systems stronger than resolution. In *23rd International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 166–181, Cham, 2020.

12 Maria Luisa Bonet, Jordi Levy, and Felip Manyà. Resolution for max-SAT. *Artif. Intell.*, 171(8-9):606–618, 2007.

13 Sam Buss and Neil Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, Volume 17, Issue 2, April 2021. Conference version appeared in SAT'19.

14 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

15 Federico Heras and Javier Larrosa. New inference rules for efficient Max-SAT solving. In *21st National Conference on Artificial Intelligence and 18th Innovative Applications of Artificial Intelligence Conference*, pages 68–73, 2006.

16 Marijn J. H. Heule and Armin Biere. What a difference a variable makes. In *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 75–92, 2018.

17 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188, 2013.

18 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *24th International Conference on Automated Deduction (CADE)*, pages 345–359, 2013.

19 Marijn J. H. Heule, Warren A. Hunt Jr., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *25th International Conference on Automated Deduction (CADE)*, pages 591–606, 2015.

20 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64(3):533–554, 2019. Conference version appeared in CADE'17. doi:10.1007/s10817-019-09516-0.

21 Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. PRuning through satisfaction. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference (HVC)*, pages 179–194, 2017.

22 Hannes Ihalainen, Jeremias Berg, and Matti Järvisalo. Clause redundancy and preprocessing in maximum satisfiability. In *Automated Reasoning*, pages 75–94. Springer International Publishing, 2022.

23 Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing rules. In *6th International Joint Conference on Automated Reasoning (IJCAR)*, pages 355–270, 2012.

24 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn J. H. Heule. Extended resolution simulates DRAT. In *6th International Joint Conference on Automated Reasoning (IJCAR)*, pages 516–531, 2018.

25 Benjamin Kiesl, Martina Seidl, Hans Tompits, and Armin Biere. Super-blocked clauses. In *8th International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pages 45–61. Springer, 2016. doi:10.1007/978-3-319-40229-1_5.

26 Oliver Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

27 Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 903–927. IOS Press, 2021. doi:10.3233/FAIA201007.

28 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for MaxSAT. In *Constraint Programming (CP)*, volume 210, pages 38:1–38:18, 2021.

29 Adrián Rebola-Pardo and Martin Suda. A theory of satisfiability-preserving proofs in SAT solving. In *22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 583–603, 2018.

30 Emre Yolcu and Marijn J. H. Heule. Exponential separations using guarded extension variables. In Yael Tauman Kalai, editor, *14th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 251 of *LIPIcs*, pages 101:1–101:22, 2023. doi:10.4230/LIPICS.ITCS.2023.101.

Enhancing MaxSAT Local Search via a Unified Soft Clause Weighting Scheme

Yi Chu
Institute of Software, Chinese Academy of Sciences, Beijing, China

Chu-Min Li
MIS, UR 4290, Université de Picardie Jules Verne, Amiens, France
Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Furong Ye
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Shaowei Cai[1]
Key Laboratory of System Software, Chinese Academy of Sciences, Beijing, China
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract

Local search has been widely applied to solve the well-known (weighted) partial MaxSAT problem, significantly influencing many real-world applications. The main difficulty to overcome when designing a local search algorithm is that it can easily fall into local optima. Clause weighting is a beneficial technique that dynamically adjusts the landscape of search space to help the algorithm escape from local optima. Existing works tend to increase the weights of falsified clauses, and such strategies may result in an unpredictable landscape of search space during the optimization process. Therefore, in this paper, we propose a Unified Soft Clause Weighting Scheme called Unified-SW, which increases the weights of all soft clauses in feasible local optima, whether they are satisfied or not, while preserving the hierarchy among them. We implemented Unified-SW in a new local search solver called *USW-LS*. Experimental results demonstrate that *USW-LS*, outperforms the state-of-the-art local search solvers across benchmarks from anytime tracks of recent MaxSAT Evaluations. More promisingly, a hybrid solver combining *USW-LS* and TT-Open-WBO-Inc won all four categories in the anytime track of MaxSAT Evaluation 2023.

2012 ACM Subject Classification Theory of computation → Randomized local search

Keywords and phrases Weighted Partial MaxSAT, Local Search Method, Weighting Scheme

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.8

Supplementary Material *Software (Source Code)*: `https://github.com/filyouzicha/USW-LS`
archived at `swh:1:dir:d24759a3c14cbec511b97300cf55409c9f6cc9bf`

Funding This work was supported by the National Natural Science Foundation of China under Grant 62302492, and the French Agence Nationale de la Recherche, via AI chair ANR-19-CHIA-0013-01 (Massal'IA).

1 Introduction

Given a propositional formula in conjunctive normal form (CNF), the maximum satisfiability problem (MaxSAT) aims to find an assignment that maximizes the number of satisfied clauses. Nowadays, research on MaxSAT mostly focuses on the partial MaxSAT (PMS), in

[1] Corresponding author.

which clauses are divided into hard and soft ones, and a feasible assignment, i.e., solution, requires all hard clauses to be satisfied, and an optimal solution should maximize the number of satisfied soft clauses. PMS can be generalized to weighted PMS (WPMS) by assigning positive integer weights to soft clauses to establish a hierarchy of importance among them, and the objective is to find an assignment that satisfies all hard clauses and maximizes the sum of weights of satisfied soft clauses. Note that when all weights are identical, WPMS becomes PMS. Therefore, we use WPMS to represent both WPMS and PMS. By focusing on WPMS, we are focusing on both WPMS and PMS.

In addition to obtaining much attention in computer science theory, WPMS has also been studied well in practical application domains since it can be applied to solve many real-world optimization problems such as scheduling [20], the maximum clique problem [32], FPGA routing [21], and computational protein design [2]. State-of-the-art WPMS solvers can be either exact or anytime. Exact solvers are guaranteed to find an optimal solution, and anytime solvers can quickly find a solution. With decades of development, Exact and anytime solvers have both achieved success in solving WPMS, and they are highly complementary. Recent annual MaxSAT Evaluations[2] include two main tracks. One is to assess exact solvers, and the other is to assess anytime solvers with the WPMS instances that no exact solver can solve within 1 hour. Exact solvers mainly comprise two categories: SAT-based [4, 19, 25, 5, 44, 35, 36, 6, 8, 9] and Branch-and-Bound (BnB) [31, 22, 27, 26, 1, 16, 30, 33]. Leading anytime solvers are hybrid [17, 43, 48, 29]: they apply LS for preprocessing and then resort to SAT-based methods. This paper focuses on the LS component of solvers for WPMS.

The LS method for WPMS iteratively chooses a variable and flips its value (from *True* to *False*, or from *False* to *True*) following a greedy strategy to satisfy all hard clauses and maximize the sum of weights of satisfied soft clauses. The main difficulty is that it can frequently fall into local optima, which means that flipping any variable cannot increase the number of satisfied hard clauses or the sum of weights of satisfied soft clauses. Many approaches have been introduced to overcome this difficulty, among which perhaps the most successful is *clause weighting* that is used in recent state-of-the-art LS solvers, such as SATLike [28], NuWLS [18], and BandMaxSAT [50], etc. In practice, a clause weighting scheme favors increasing the weights of falsified clauses so that their variables have greater chances to be flipped subsequently to satisfy them.

While various heuristics have been proposed for designing weighting schemes [15, 46, 45, 14, 28, 18], existing work faces a key issue: *the landscape of search space is blindly adjusted with clause weighting because the relations among the soft clauses, as well as the relations between hard clauses and soft clauses, are disturbed,* and solvers have to search for better solutions in a blindly adjusted search space, so that they can easily be misled when choosing the next variable to flip.

In this paper, we carefully analyze the questions to answer when designing a clause weighting scheme and propose a novel weighting scheme named Unified-SW (Unified Soft Clause Weighting), which distinguishes feasible local optima from infeasible local optima, and increases the weights of all soft clauses in feasible local optima, whether they are satisfied or not, while preserving the hierarchy among them, which is quite different from some previous approaches which tend to only increase the weights of falsified clauses.

Based on Unified-SW, we develop a new LS solver named *USW-LS* for WPMS. We compare *USW-LS* with state-of-the-art LS solvers on unweighted and weighted benchmarks from the anytime tracks of MaxSAT Evaluations from 2018 to 2023. Experimental results

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

demonstrate that *USW-LS* outperforms the competing solvers in terms of both fixed-budget performance and anytime performance. In addition, *USW-LS* considerably advances the state-of-the-art in WPMS solving. Notably, the hybrid solver that combines *USW-LS* and TT-Open-WBO-Inc [41] has won all four categories in the anytime track of MaxSAT Evaluation 2023.

The rest of the paper is organized as follows. Section 2 presents some notions used in the paper. Section 3 presents related works. Section 4 analyzes the questions to answer when designing a clause weighting scheme, describes Unified-SW and its distinguishing properties, and presents our new LS solver *USW-LS* based on Unified-SW. Section 5 empirically demonstrates the performance of *USW-LS* and the effectiveness of Unified-SW. Section 6 concludes.

2 Preliminaries

Consider a set of n Boolean variables, denoted as $\{x_1, x_2, \cdots, x_n\}$. Corresponding to these variables is the set of $2n$ literals, defined as $\{x_1, \neg x_1, x_2, \neg x_2, \cdots, x_n, \neg x_n\}$, where each literal represents either a variable or its negation. A clause c of length k is a disjunction of k literals, denoted as $c = l_1 \vee l_2, \cdots, \vee l_k$. A propositional formula F in Conjunctive Normal Form (CNF) is a conjunction of clauses, denoted as $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$. For a CNF formula F, a complete assignment is a mapping that assigns each variable in F to a Boolean value (*True* or *False*). In this paper, an assignment is always complete. A literal x ($\neg x$) is satisfied if x is assigned *True* (*False*), otherwise, it is falsified. Each clause has two states: satisfied or falsified. Given an assignment α, a clause c is satisfied if at least one literal in c is *True*; otherwise, c is falsified.

PMS divides clauses in a CNF formula into hard and soft ones, and its objective is to find an assignment that maximizes the number of satisfied soft clauses while satisfying all hard clauses. For WPMS, each soft clause has a positive integer weight representing a cost if the soft clause is falsified, so that there is a hierarchy of importance among soft clauses, and the goal is to find an assignment that maximizes the total weight of satisfied soft clauses while satisfying all hard clauses. PMS represents a special case of WPMS in which the weight of each soft clause is identical, represented by 1.

An assignment is feasible if it satisfies all hard clauses. A feasible assignment is also called a *solution*. The cost of an assignment α is the sum of weights of falsified soft clauses, denoted as $cost(\alpha)$. An optimal solution is obtained by minimizing its cost while satisfying all hard clauses. A solution α_1 is considered to be better than another solution α_2, if $cost(\alpha_1) < cost(\alpha_2)$.

Clause weighting schemes assign weights to both hard and soft clauses, and maintain them. These weights are different from the original weights of soft clauses in WPMS instances. To avoid confusion, we will call the weights assigned and maintained by a clause weighting scheme *maintenance weights* or simply *weights*, but systematically add the word "original" when talking about the original weights of soft clauses in WPMS instances. The maintenance weight of a (hard or soft) clause c is denoted $w(c)$, and the original weight of a soft clause is denoted $w_{ori}(c)$. Note that $w_{ori}(c)$ is never changed, differently from $w(c)$. The average original weight of all soft clauses is denoted $avgw_{soft}$.

The maintenance weights are used to calculate the score of each variable. The score of variable x, denoted $score(x)$, is calculated as $score(x) = make(x) - break(x)$, where $make(x)$ is the sum of maintenance weights of (hard and soft) clauses changing from falsified to satisfied upon flipping x, and $break(x)$ is the sum of maintenance weights of (hard and soft) clauses changing from satisfied to falsified upon flipping x.

Restart strategy has commonly been applied when applying LS for WPMS. When restarting a local search, apart from initializing the current assignment, the maintenance weights assigned to each clause may also be reset.

3 Related Work

Solvers for WPMS can be categorized into exact and anytime types. Exact solvers can prove the optimality of their solutions when the algorithm terminates, and these methods have made significant progress over decades. In particular, the SAT-based solvers, which are built based on classic SAT solvers, have gone through a series of advancements [21, 3, 4, 19, 25, 5, 44, 35, 36, 6, 9]. BnB MaxSAT solvers [22, 27, 26, 1, 16, 33] implement the branch-and-bound method, detecting inconsistent subsets of soft clauses through unit propagation and lower bound computation. The performance of the BnB solvers has been significantly enhanced by integrating clause learning and an efficient bounding procedure [33].

While unable to prove solution optimality, LS solvers often obtain high-quality solutions within short runtimes and exhibit complementarity with exact solvers when solving various types of WPMS instances. Early LS algorithms encoded PMS instances as weighted MaxSAT instances, which were then solved accordingly [23].

LS algorithms for solving WPMS have been developed along a line of using clause weighting schemes to guide search. These algorithms assign weights to clauses and calculate the score of variables using these weights. So, they intensify or diversify the search primarily by maintaining these assigned weights in different ways.

Note that in PMS, as well as in WPMS, a hard clause is definitely more important than all soft clauses. So, a natural way to assign weights to clauses is to set the weight of each hard clause larger than the sum of all soft clause weights. However, this approach biases the search towards satisfying hard clauses, significantly limiting the search space and often negatively affecting the performance of LS algorithms [15].

As the above natural way to assign weights to clauses is ineffective, researchers spent much effort designing better clause weighting schemes for WPMS for a long time. Examples of early clause weighting schemes can be found in [15] for statically assigning suitable weights to hard clauses or in [46, 45] for dynamically adjusting hard clause weights during the search.

Below, we go over recent clause weighting schemes, included in the state-of-the-art algorithms Dist [14], SATLike [28] and NuWLS [18].

Dist uses a weighting scheme that only updates the hard clause weights. It introduces the concepts of hard score and soft score and employs different heuristic methods for hard clauses and soft clauses. The Dist series algorithms [13, 34] greatly improve the performance compared to previous LS algorithms for WPMS. However, the concepts and heuristic methods are separately defined for hard and soft clauses, increasing the algorithm complexity. Moreover, the algorithms prioritize satisfying hard clauses, limiting the search space. Subsequently, a series of algorithms based on Dist was designed to exploit the structure of WPMS further.

SATLike [28] introduces a weighting scheme called Weighting-PMS, which utilizes different increments to update the weights of hard and soft clauses, and sets a uniform upper bound for all soft clause weights. Weighting-PMS is also used in SATLike3.0 [12] and BandMaxSAT [50].

Based on an empirical finding that initial soft clause weights have a clear effect on the effectiveness of the algorithm adopting Weighting-PMS, NuWLS [18] uses a new weighting scheme named Dist-Weighting. There are three main distinctions between Dist-Weighting and previous weighting schemes. First, Dist-Weighting initializes soft clause weights within a reasonable range. Second, it distinguishes conditions for updating hard and soft clause weights. Third, it associates a specific upper bound with each soft clause based on its original weight, instead of setting it uniformly as in Weighting-PMS.

Unfortunately, despite the above progress, designing effective clause weighting schemes in LS algorithms for WPMS remains challenging, because it is difficult to balance the weight relations between hard and soft clauses, as well as within soft clauses. After all, the initial or adjusted weights may fail to reflect the importance of hard or soft clauses as expressed by their hardness or original weights, potentially causing significant disturbance to the search space. Moreover, while falsifying a hard clause is necessarily an error that should be repaired by increasing its weight to favor its satisfaction subsequently, falsifying a soft clause is not necessarily an error because it can be falsified in an optimal solution. So, it is unclear how to distinguish between satisfied and falsified soft clauses by adjusting their weight.

Anytime solvers can employ exact methods to provide the best real-time solution. Loandra [10] and TT-Open-WBO-Inc [38, 40] (based on Open-WBO-Inc [24]), are two notable instances of such anytime solvers. The complementarity between exact and LS methods has led to another research line that combines these two methods, giving rise to hybrid solvers. Typically, these hybrid solvers apply an SAT solver to obtain a feasible assignment. Then, LS is executed, using this assignment as the initial assignment, until no improvement for k consecutive steps or a short time budget is reached. The best solution found by LS will serve as an initial model and provide an initial upper bound for an SAT-based component. This hybrid-solving process will continue until the total budget is exhausted. For the weighted part of the SAT-based component, the best-performing anytime solver TT-Open-WBO-Inc [40] combines weights approximation by Boolean Multilevel Optimization [24] and the SAT-based LS algorithm Polosat [39]; for its SAT-based unweighted component, it combines Mrs. Beaver algorithm [37] and Polosat.

Hybrid solvers demonstrate impressive performance in the anytime tracks of recent MaxSAT Evaluations (MSEs). Hybrid solvers won 2 out of 4 anytime categories at MSE 2020, 3 out of 4 anytime categories at MSE 2021, and all 4 anytime categories at the two latest evaluations, MSE 2022 and MSE 2023.

4 Proposed Methodology

In this section, we first analyze the questions to answer when designing a clause weighting scheme, then propose a novel clause weighting scheme called Unified Soft Clause Weighting Scheme (Unified-SW) to provide simple and effective responses to these questions, based on which a new LS algorithm for WPMS called *USW-LS* is designed.

4.1 Analysis of clause weighting scheme

Recent LS solvers for WPMS consist of iteratively choosing the next variable x to flip, based on its score $score(x)$. Recall that $score(x) = make(x) - break(x)$, where $make(x)$ ($break(x)$) is the sum of weights of soft and hard clauses that will become satisfied (falsified) upon flipping x. The weight of each clause is initialized at the beginning of the search process and dynamically adjusted later on. Note that no distinction is made between hard and soft clauses when calculating $score(x)$ once their weights are defined. This working scheme raises the following questions to answer when designing an LS solver:

- How to initialize and adjust the weight of each hard clause and each soft clause to distinguish between them during the search? In particular, what is the increment to be added to the weight of a soft clause?

- Can the weight of a clause be infinitely increased when the search proceeds? In other words, should there be an upper bound for the weight of a clause? If yes, what is the upper bound? Should it be uniform for all clauses or specific for each clause?
- When the search encounters a local optimum, how to select the clauses whose weights should be adjusted?

Different responses to these questions yield solvers of different performances. State-of-the-art LS solvers either perform prior extensive experimental analysis or use intuition to answer the first two questions, and the responses are often instance-type specific and are hard to obtain for new instance types. In fact, the difficulty in designing an LS solver is that a very small change in the solver can considerably deteriorate its performance.

Furthermore, existing weighting schemes usually increase the weights of falsified clauses and leave the weights of other clauses unchanged to answer the last question. This is reasonable for a falsified hard clause because a falsified hard clause represents an error that must be repaired. However, the situation is much more complicated for a falsified soft clause, because falsifying a soft clause is not necessarily an error. Moreover, only increasing the weights of falsified soft clauses while leaving the weights of satisfied soft clauses unchanged may change the hierarchy of soft clauses to mislead the solver. In other words, if the original weight of a soft clause c_1 is greater than another soft clause c_2, i.e., if c_1 is originally considered to be more important than c_2, increasing the weight of c_2 but leaving the weight of c_1 unchanged may make the solver satisfy c_2 in priority so that the solver becomes further away from the optimal solution.

In the next subsection, we propose a novel clause weighting scheme that provides simple and effective responses to all the above questions.

4.2 Unified-SW: a novel weighting scheme

We propose the Unified-SW scheme, where Unified-SW stands for unified soft clause weighting. It operates as follows:

Initialization of Clause Weights: At the beginning of each round (restart) of local search, Unified-SW initializes each clause weight as follows:
  - For each hard clause c, the weight $w(c) := 1$.
  - For each soft clause c, the weight $w(c) := 0$.

Update of Clause Weights: When the search encounters a local optimum α, the clause weights are updated as follows:
  - If α falsifies at least one hard clause, and flipping any variable cannot increase the total weight of satisfied hard clauses, α is called an *infeasible local optimum*. In this case, for each falsified hard clause c, $w(c) := w(c) + 1$.
  - Otherwise, α is a *feasible local optimum* (i.e., all hard clauses are satisfied, but α cannot be improved by flipping any variable). Let k be the number of feasible local optima encountered so far. For each soft clause c, $w(c) := k \times \frac{w_{ori}(c)}{avgw_{soft}}$, where w_{ori} is the original weight of c in the WPMS instance and $avgw_{soft}$ is the average original weight of all soft clauses.

Note that while the original weights of soft clauses are positive integers, the weights defined by Unified-SW are positive real numbers with double precision.

Unified-SW obtains the following distinguishing properties.

- At the beginning of the search, the soft clauses are not considered in the score of any variables because their weight is 0. Consequently, the solver focuses on searching for the first solution by increasing the weight of the hard clauses, and the first found solution is necessarily a feasible local optimum.

- At each feasible local optimum, the weight of *each* soft clause is increased proportionally to its original weight, whether it is satisfied or not, which significantly differs from the previous approaches in which only the weight of falsified soft clauses is increased. It is for this reason that the new clause weighting scheme is called Unified-SW. The intuition behind this unified soft clause weighting is that even a satisfied soft clause in a feasible local optimum can represent an error that should be repaired.
- The hierarchy between soft clauses is always kept, i.e., given any two soft clauses c_1 and c_2, if $w_{ori}(c_1) \leq w_{ori}(c_2)$, then $w(c_1) \leq w(c_2)$ for any k. Moreover, the difference of weights between c_1 and c_2 is $k \times (\frac{w_{ori}(c_1)}{avgw_{soft}} - \frac{w_{ori}(c_2)}{avgw_{soft}})$ which will become larger and larger when search proceeds. In other words, the more the solver encounters feasible local optima, the more important the soft clauses with great original weight will be, so that it will be considered a priority to improve the cost of a solution.
- Let $S = \{c_1, \ldots, c_s\}$ be the set of all soft clauses. After encountering k^{th} feasible local optima, the total weight of soft clauses is $k \times s$. In other words, the more the solver encounters feasible local optima, the larger the total weight of soft clauses. This is relevant because a feasible local optimum is often dominated by hard clauses, i.e., a feasible local optimum is often encountered by satisfying hard clauses. However, when there are many feasible local optima, soft clauses should be considered more in the score of variables to improve the cost of the solutions.
- The weights of hard and soft clauses are increased upon infeasible and feasible local optima, respectively, and infeasible local optima are considered in priority. The increase of weights of soft clauses upon a feasible local optimum can make the solver far from solutions, and then the solver works with the new greater weights of soft clauses to hopefully move again toward a better solution via new paths, eventually by increasing the weights of hard clauses in infeasible local optima along the new paths.
- There is no upper bound, neither for the weights of soft clauses nor for the weights of hard clauses, which greatly simplifies algorithm design.
- The best solution found so far in terms of weights defined by Unified-SW is also the best solution found so far in terms of original weights. To see this, we look at k and denote the best solution found so far by α^*. When $k = 0$, i.e., at the beginning, the first solution found is obviously the best solution found so far. When $k > 0$, let c denote a falsified soft clause, $w(c) = \sum_c k \times \frac{w_{ori}(c)}{avgw_{soft}}$ which is the smallest if and only if $cost(\alpha^*) = \sum_c w_{ori}(c)$ is the smallest. Therefore, when a solver finds the best solution so far in terms of weights defined by Unified-SW, it does not need to test if it is also the best solution found so far in terms of original weights.

Note that for PMS, which is a particular case of WPMS where both $w_{ori}(c)$ and $avgw_{soft}$ are 1 for any soft clause c, the weight of each soft clause is increased by 1 in each feasible local optimum, increasing the importance of the soft clauses w.r.t. hard clauses and hopefully making the next solution better.

4.3 The *USW-LS* algorithm

Based on the above Unified-SW, we introduce a new LS algorithm named *USW-LS*. The pseudo-code of *USW-LS* is outlined in Algorithm 1. We use α^* and $cost^*$ to denote the best-found solution so far and the *cost* value of the best-found solution, respectively, while α represents the current assignment maintained during the search.

In the *USW-LS* algorithm, α^* is initialized as empty, and $cost^*$ is initialized to positive infinity. It then iteratively executes the local search process until a termination criterion is met (Lines 2–20). A round is from line 3 to line 20.

Algorithm 1 *USW-LS*.

```
Input: WPMS instance $F$, cutoff time.
Output: The best-found solution and its cost, or "No solution found".
1  $\alpha^* := \emptyset$; $cost^* := +\infty$;
2  while no terminating criteria are met do
3      $\alpha :=$ an initial complete assignment;
4      Initialize clause weights by Unified-SW;
5      $L = 10\,000\,000$;
6      for $step = 0$; $step < L$; $step{+}{+}$ do
7          if $\alpha$ is feasible and $cost^* > cost(\alpha)$ then
8              $\alpha^* := \alpha$; $cost^* := cost(\alpha)$; $L = step + 10^7$;
9              if $cost^* == 0$ then
10                 return $\alpha^*$ and $cost^*$;
11         if $(D : \{x|score(x) > 0\}) \neq \emptyset$ then
12             $v :=$ a variable in $D$ selected by the BMS strategy;
13         else
14             update clause weights by Unified-SW;
15             if $\exists$ falsified hard clauses then
16                 $c :=$ a random falsified hard clause;
17             else
18                 $c :=$ a random falsified soft clause;
19             $v :=$ the variable with highest score in $c$;
20         $\alpha := \alpha$ with $v$ flipped;
21 if $\alpha^* \neq \emptyset$ then return $\alpha^*$ and $cost^*$;
22 else return "No solution found";
```

In the local search process, an initial assignment is generated by a unit propagation based procedure (Line 3) [12]. *USW-LS* then initializes the weights of all clauses by the Unified-SW scheme. After initialization, *USW-LS* conducts the search process (Lines 6–20). During the search process, whenever *USW-LS* finds a solution whose cost value is lower than $cost^*$, α^* and $cost^*$ are updated accordingly.

In each search step, let D denote the set of variables with $score(x)>0$. *USW-LS* selects a variable and flips its value depending on two situations. (I) If D is not empty, a variable is selected from D. Since traversing all elements in D would be time-consuming, a sampling strategy called BMS (Best from Multiple Selections) [11] is adopted here. Through BMS, t variables are randomly selected from the D set (Where t is a parameter in BMS, the value of which is specified in the Experimental setup.), and then the variable with the highest score among the t variables is chosen to flip. It is shown in [11] that the score of the variable selected in this way is close to the highest in D with high probability. (II) If D is empty, indicating that the search is stuck in a local optimum, *USW-LS* updates the weights of clauses according to Unified-SW. Then *USW-LS* randomly selects a falsified clause c, and picks the variable with the highest score from the selected clause c.

Finally, when any terminating criterion is met, *USW-LS* reports α^* and $cost^*$ if a solution is found; otherwise, it reports "No solution found".

5 Experimental Evaluations

In this section, we introduce experimental preliminaries and then conduct extensive experiments on unweighted and weighted benchmarks from the anytime tracks of MaxSAT Evaluations (MSEs) from 2018 to 2023. First, we compare *USW-LS* with three state-of-the-art LS solvers. Second, we demonstrate the performance enhancement achieved by combining *USW-LS* with an SAT-based solver. Third, we present experimental results to demonstrate the anytime performance exhibited by LS solvers employing different weighting schemes. Finally, we examine the effectiveness of uniformly adjusting soft clause weights.

5.1 Experimental preliminaries

Benchmarks. Our experiments are conducted on 12 benchmarks, i.e., unweighted and weighted benchmarks from the anytime tracks of MSEs from 2018 to 2023.

Competitors. In the first experiment, we compare *USW-LS* with the following three state-of-the-art LS solvers, all of which employ various weighting schemes. The source codes of these three solvers are publicly available,[3,4,5] and the parameter settings follow those presented in their papers.

- NuWLS [18], which uses the Dist-Weighting scheme.
- BandMaxSAT [50], which adopts the Weighting-PMS scheme proposed in SATLike [28]
- SATLike3.0 [12], which adopts the Weighting-PMS scheme.

In the second experiment, we combine *USW-LS* with TT-Open-WBO-Inc (MSE2020 version) [40], which is based on Open-WBO-Inc [24], resulting in a new hybrid solver named *USW-LS-c*.

First, we compare *USW-LS-c* with NuWLS-c, DT-HyWalk, and SATLike-c.

- NuWLS-c combines NuWLS with TT-Open-WBO-Inc (MSE2020 version). We use its source code from MSE 2022.[6]
- DT-HyWalk combines BandMaxSAT with other LS and TT-Open-WBO-Inc (MSE2020 version). We use its source code from MSE 2022.[6]
- SATLike-c combines SATLike with TT-Open-WBO-Inc (MSE2020 version). We use its source code from MSE 2021.[7]

Then, we compare *USW-LS-c* with all hybrid solvers from the anytime track of MSE 2023.[8]

- TT-Open-WBO-Inc-23(G) [43] combines NuWLS with TT-Open-WBO-Inc, where *Glucose4.1* [7] serves as the underlying SAT solver.
- TT-Open-WBO-Inc-23(I) [43] combines NuWLS with TT-Open-WBO-Inc, where *IntelSAT* [42] serves as the underlying SAT solver.
- NuWLS-c-Band [48] combines NuWLS with BandMaxSAT and TT-Open-WBO-Inc.
- NuWLS-c-FPS [48] combines NuWLS with the farsighted probabilistic sampling (FPS) strategy [49] and TT-Open-WBO-Inc.

The purpose of the second experiment is to investigate whether *USW-LS* could improve the performance of hybrid solvers by combining LS with SAT-based algorithms. Note that this experiment does not include NuWLS-c-2023 [17] because NuWLS-c-2023 is identical to *USW-LS-c*, combining *USW-LS* with TT-Open-WBO-Inc.

[3] `https://github.com/filyouzicha/NuWLS`
[4] `https://github.com/JHL-HUST/BandMaxSAT`
[5] `http://lcs.ios.ac.cn/%7ecaisw/MaxSAT.html`
[6] `https://maxsat-evaluations.github.io/2022`
[7] `https://maxsat-evaluations.github.io/2021`
[8] `https://maxsat-evaluations.github.io/2023`

In the third experiment, we analyze the anytime performance of *USW-LS* against NuWLS, BandMaxSAT, and SATLike3.0.

In the fourth experiment, we make an ablation study to compare *USW-LS* with a variant identical to *USW-LS* except that it only increases the weights of the falsified soft clauses in a feasible local optimum.

Experimental setup. Our *USW-LS* solver is implemented in C++. *USW-LS* employs the BMS strategy with a parameter t denoting the sample count, following the settings of NuWLS. i.e., for PMS, t is set to 96; for WPMS, t is set to 25.

All solvers are compiled with g++ using the "-O3" option. The experiments are conducted on a workstation running Ubuntu (version=" 20.04.4 LTS (Focal Fossa)") and equipped with AMD EPYC 7763 3.2GHz CPUs.

Consistent with the rules in the anytime tracks of recent MSEs, we employ two cutoff times, 60 seconds and 300 seconds. Each solver performs one run within a given cutoff time on each instance. We record the cost of the best solution found by solver S on instance I, denoted as $cost_{SI}$. The cost of the best solution found among all solvers in the same table within the same cutoff time on instance I is denoted as $cost_{bI}$. The cost of the best-known solution on instance I is denoted as $best_I$. For each solver S solving a benchmark B within a cutoff time, we use two metrics to evaluate the performance of S.

- $\#win$: the number of instances where the corresponding $cost_{bI}$ can be obtained by solver S on B (i.e., the number of instances on which S wins). The number of winning instances is a metric widely used to evaluate the performance of LS WPMS solvers.
- avg_{score}: we use $score_{SI}$ to denote the competition score of solver S on instance I, if S could not report any solution on instance I within the cutoff time, then $score_{SI} = 0$. Otherwise, $score_{SI} = \frac{best_I+1}{cost_{SI}+1}$. We use avg_{score} to denote the average competition score of a solver on a benchmark. The competition score is the metric to measure the performance of anytime solvers in recent MSEs.

The number of instances in each benchmark is indicated by '#inst'. For each of the above two metrics, if a solver obtains a larger metric value on a benchmark, then the solver exhibits better performance on the benchmark. The results highlighted in **bold** indicate the best performance for the corresponding metric.

5.2 Comparison with local search solvers

The comparative results of *USW-LS* and its LS competitors on all the benchmarks are shown in Table 1. On all the unweighted and weighted benchmarks, for both 60-second and 300-second cutoff times, *USW-LS* outperforms all competing solvers in terms of the number of winning instances ($\#win$) and average scores (avg_{score}).

- For unweighted benchmarks with a 60-second cutoff time, *USW-LS* outperforms the second-ranked solver by 28.40-58.57% for $\#win$ and 1.54-6.51% for avg_{score}. With a 300-second cutoff time, *USW-LS* outperforms the second-ranked solver by 31.48- 63.64% for $\#win$ and 1.33-8.19% for avg_{score}.
- For weighted benchmarks with a 60-second cutoff time, *USW-LS* exceeds the second-ranked solver by 45.07-157.58% for $\#win$, and 1.34-7.74% for avg_{score}. With a 300-second cutoff time, *USW-LS* exceeds the second-ranked solver by 65.29-127.08% for $\#win$ and 0.98-6.97% for avg_{score}.

Among these LS solvers, the only distinction between *USW-LS*, NuWLS, and SATLike3.0 lies in the weighting scheme utilized. Both BandMaxSAT and SATLike3.0 employ the same weighting scheme, differing only in the strategy introduced by BandMaxSAT for variable

Table 1 Comparisons of *USW-LS* with state-of-the-art LS solvers.

Benchmark	#inst	*USW-LS*		NuWLS		BandMaxSAT		SATLike3.0	
		$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}
Unweighted (60 seconds)									
Unw_18	153	**102**	**0.7426**	69	0.7131	42	0.6455	45	0.5673
Unw_19	299	**206**	**0.7297**	151	0.7117	96	0.6575	106	0.6179
Unw_20	262	**181**	**0.7360**	127	0.7161	76	0.6679	77	0.6150
Unw_21	155	**104**	**0.6578**	81	0.6176	52	0.5778	43	0.5223
Unw_22	179	**113**	**0.7092**	78	0.6684	55	0.6287	37	0.5682
Unw_23	179	**111**	**0.6629**	70	0.6528	30	0.5738	21	0.5160
Unweighted (300 seconds)									
Unw_18	153	**109**	**0.7726**	71	0.7479	54	0.6906	46	0.6113
Unw_19	299	**213**	**0.7546**	162	0.7447	116	0.6961	116	0.6506
Unw_20	262	**187**	**0.7616**	132	0.7435	95	0.6965	90	0.6430
Unw_21	155	**111**	**0.6877**	79	0.6419	59	0.6013	46	0.5495
Unw_22	179	**126**	**0.7514**	77	0.6945	65	0.6540	44	0.5955
Unw_23	179	**111**	**0.6977**	71	0.6881	39	0.6219	34	0.5721
Weighted (60 seconds)									
Wei_18	172	**103**	**0.7716**	71	0.7614	24	0.6978	18	0.6809
Wei_19	297	**192**	**0.7686**	125	0.7419	65	0.6638	41	0.6581
Wei_20	253	**162**	**0.7797**	94	0.7436	43	0.6301	36	0.6391
Wei_21	151	**75**	**0.6833**	38	0.6343	24	0.5437	27	0.5740
Wei_22	197	**105**	**0.7094**	58	0.6751	35	0.6052	25	0.6095
Wei_23	160	**85**	**0.6631**	33	0.6182	31	0.5651	15	0.5671
Weighted (300 seconds)									
Wei_18	172	**111**	**0.7848**	67	0.7772	27	0.7244	16	0.7042
Wei_19	297	**200**	**0.7916**	121	0.7715	81	0.7118	54	0.7034
Wei_20	253	**170**	**0.8085**	99	0.7784	54	0.6893	39	0.6815
Wei_21	151	**74**	**0.7199**	35	0.6817	37	0.6225	28	0.6299
Wei_22	197	**109**	**0.7459**	57	0.7124	48	0.6705	28	0.6629
Wei_23	160	**83**	**0.7093**	37	0.6630	34	0.6338	16	0.6293

selection after encountering local optima. Experimental results demonstrate that *USW-LS* outperforms NuWLS, while NuWLS outperforms SATLike3.0 in terms of $\#win$ and avg_{score}, indicating that a well-designed weighting scheme can significantly enhance the performance of LS solvers.

5.3 Improving hybrid solver through *USW-LS*

Since our experimental results in Table 1 demonstrate that *USW-LS* performs much better than state-of-the-art LS solvers for solving WPMS on all benchmarks, we are interested in investigating whether *USW-LS* could improve the performance of hybrid solvers compared to these LS solvers. Therefore, we compare *USW-LS-c* with NuWLS-c, DT-HyWalk, and SATLike-c, and the experimental results are shown in Table 2. From Table 2, in terms of $\#win$ and avg_{score}, our *USW-LS-c* solver outperforms all the competitors on all benchmarks and runtimes, indicating that *USW-LS* can considerably advance the performance of hybrid solvers.

We submitted *USW-LS-c* to the anytime track of MSE 2023, where *USW-LS-c* is referred to as NuWLS-c-2023. The official results from the anytime track of MSE 2023 indicate that the top-five solvers are all hybrid, and USW-LS-c *won all four categories*.

Table 2 Comparisons of *USW-LS-c* with hybrid solvers NuWLS-c, DT-HyWalk, and SATLike-c.

Benchmark	#inst	*USW-LS-c*		NuWLS-c		DT-HyWalk		SATLike-c	
		#win	avg_{score}	*#win*	avg_{score}	*#win*	avg_{score}	*#win*	avg_{score}
`Unweighted` (60 seconds)									
`Unw_18`	153	**106**	**0.8239**	74	0.8015	83	0.7960	52	0.7745
`Unw_19`	299	**234**	**0.8717**	175	0.8549	189	0.8349	158	0.8265
`Unw_20`	262	**202**	**0.8606**	138	0.8420	151	0.8257	122	0.8214
`Unw_21`	155	**115**	**0.8280**	86	0.8011	87	0.7870	80	0.7834
`Unw_22`	179	**128**	**0.8239**	75	0.7896	89	0.7776	72	0.7610
`Unw_23`	179	**109**	**0.7832**	73	0.7605	75	0.7268	66	0.7220
`Unweighted` (300 seconds)									
`Unw_18`	153	**98**	**0.8635**	77	0.8501	87	0.8507	52	0.8249
`Unw_19`	299	**228**	**0.9141**	186	0.9031	204	0.8950	160	0.8806
`Unw_20`	262	**190**	**0.9025**	144	0.8888	161	0.8754	121	0.8635
`Unw_21`	155	**116**	**0.8864**	96	0.8752	97	0.8646	80	0.8502
`Unw_22`	179	**125**	**0.8866**	90	0.8773	102	0.8601	83	0.8482
`Unw_23`	179	**118**	**0.8609**	88	0.8490	94	0.8274	84	0.8202
`Weighted` (60 seconds)									
`Wei_18`	172	**101**	**0.8811**	84	0.8772	69	0.8706	76	0.8699
`Wei_19`	297	**185**	**0.8572**	133	0.8411	122	0.8233	123	0.8352
`Wei_20`	253	**144**	**0.8518**	110	0.8386	88	0.8006	92	0.8061
`Wei_21`	151	**75**	**0.7800**	59	0.7481	44	0.7102	46	0.7090
`Wei_22`	197	**105**	**0.7862**	66	0.7617	53	0.7325	60	0.7390
`Wei_23`	160	**89**	**0.7867**	54	0.7790	34	0.7296	44	0.7458
`Weighted` (300 seconds)									
`Wei_18`	172	**108**	**0.9147**	89	0.8985	82	0.9007	87	0.9003
`Wei_19`	297	**197**	**0.9202**	164	0.9054	153	0.9021	137	0.8962
`Wei_20`	253	**163**	**0.9178**	129	0.8949	115	0.8653	106	0.8568
`Wei_21`	151	**83**	**0.8475**	65	0.8317	53	0.7798	52	0.7516
`Wei_22`	197	**116**	**0.8676**	90	0.8516	68	0.7838	62	0.7760
`Wei_23`	160	**96**	**0.8891**	63	0.8703	45	0.8384	50	0.8313

To give a more global assessment of *USW-LS-c* against state-of-the-art anytime solvers, we compare it with the four other top-five solvers NuWLS-c-Band, NuWLS-c-FPS, TT-Open-WBO-Inc-23(G), and TT-Open-WBO-Inc-23(I) in the anytime track of MSE 2023 on all benchmarks from the anytime tracks of MSE since 2018. The experimental results are reported in Table 3. In terms of *#win*, *USW-LS-c* outperforms all the competitors on all comparisons. In terms of avg_{score}, *USW-LS-c* outperforms all the competing solvers on 18 out of 24 comparisons. On the other 6 comparisons, *USW-LS-c* ranks second and slightly worse than the best.

5.4 Anytime performance analysis

Apart from the fixed-budget performance measures that have been commonly applied for MSEs and previous work on WPMS, an anytime performance measure [47], which evaluates the performance of solvers across multiple cutoff times, has been used for assessing four WPMS LS solvers. We also take this measure into account for comparing our proposed *USW-LS* with three LS solvers.

Table 3 Comparisons of *USW-LS-c* with hybrid solvers from MSE 2023, TT-OWI-G refers to TT-Open-WBO-Inc-23(G), TT-OWI-I refers to TT-Open-WBO-Inc-23(I).

Benchmark	#inst	*USW-LS-c*		TT-OWI-G		TT-OWI-I		NuWLS-c-Band		NuWLS-c-FPS	
		$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}
`Unweighted` (60 seconds)											
`Unw_18`	153	**85**	**0.8239**	63	0.8097	58	0.8021	70	0.8092	77	0.8101
`Unw_19`	299	**200**	**0.8717**	159	0.8609	160	0.8547	170	0.8639	176	0.8623
`Unw_20`	262	**171**	**0.8606**	117	0.8445	119	0.8402	128	0.8458	139	0.8382
`Unw_21`	155	**92**	**0.8280**	77	0.8086	77	0.8093	77	0.8050	81	0.7930
`Unw_22`	179	**96**	**0.8239**	64	0.8099	63	0.8021	77	0.7921	74	0.7851
`Unw_23`	179	**89**	**0.7832**	61	0.7829	61	0.7615	66	0.7760	66	0.7770
`Unweighted` (300 seconds)											
`Unw_18`	153	**91**	**0.8635**	64	0.8519	77	0.8610	77	0.8556	77	0.8551
`Unw_19`	299	**214**	**0.9141**	163	0.9054	185	0.9108	183	0.9084	187	0.9099
`Unw_20`	262	**180**	0.9025	126	0.8913	140	**0.9035**	141	0.8891	146	0.8862
`Unw_21`	155	**105**	0.8864	86	0.8751	81	**0.8923**	84	0.8549	86	0.8508
`Unw_22`	179	**108**	**0.8866**	78	0.8779	79	0.8861	91	0.8514	88	0.8427
`Unw_23`	179	**112**	**0.8609**	81	0.8563	84	0.8568	87	0.8543	87	0.8443
`Weighted` (60 seconds)											
`Wei_18`	172	**98**	0.8811	84	**0.8816**	65	0.8713	84	0.8761	86	0.8801
`Wei_19`	297	**181**	**0.8572**	123	0.8318	122	0.8393	127	0.8406	103	0.8294
`Wei_20`	253	**140**	**0.8518**	96	0.8335	93	0.8360	95	0.8336	82	0.8291
`Wei_21`	151	**69**	**0.7800**	46	0.7502	42	0.7580	38	0.7526	44	0.7492
`Wei_22`	197	**93**	**0.7862**	67	0.7591	53	0.7542	67	0.7628	51	0.7558
`Wei_23`	160	**85**	**0.7867**	41	0.7585	35	0.7695	45	0.7755	45	0.7727
`Weighted` (300 seconds)											
`Wei_18`	172	**101**	0.9147	95	**0.9176**	71	0.9091	92	0.9062	96	0.9067
`Wei_19`	297	**186**	**0.9202**	149	0.8823	136	0.9109	144	0.8990	135	0.9000
`Wei_20`	253	**153**	**0.9178**	119	0.8926	103	0.8884	107	0.8926	107	0.8944
`Wei_21`	151	**71**	0.8475	64	**0.8550**	49	0.8459	52	0.8342	55	0.8477
`Wei_22`	197	**97**	0.8676	91	**0.8678**	68	0.8356	72	0.8350	68	0.8383
`Wei_23`	160	**90**	**0.8891**	55	0.8466	42	0.8889	49	0.8569	49	0.8686

The empirical cumulative distribution function (ECDF) is used to assess the anytime performance of solvers. Given a set of target values, i.e., cost, for each WPMS instance, an ECDF value at time t represents the proportion of targets that have been achieved by a solver within cutoff time t. Following the suggestion in the previous work [47], we consider a set of 100 cost values that are sampled within the smallest and largest cost values obtained during the optimization process of all tested solvers. We plot in Figure 1 the averaged ECDFs values across all tested WPMS instances for 100 cutoff times t that are selected within the range of $[0, 300s]$ with a logarithmic scale. We can observe that *USW-LS* generally outperforms the other tested LS solvers across all the considered cutoff times, and this advantage is significant, as shown in the figure. Note that the results are aggregated across all tested WPMS instances such that we do not observe sharp increases at a specific cutoff time or a flat line in Figure 1, but these behaviors may occur in particular instances as shown in previous work [47].

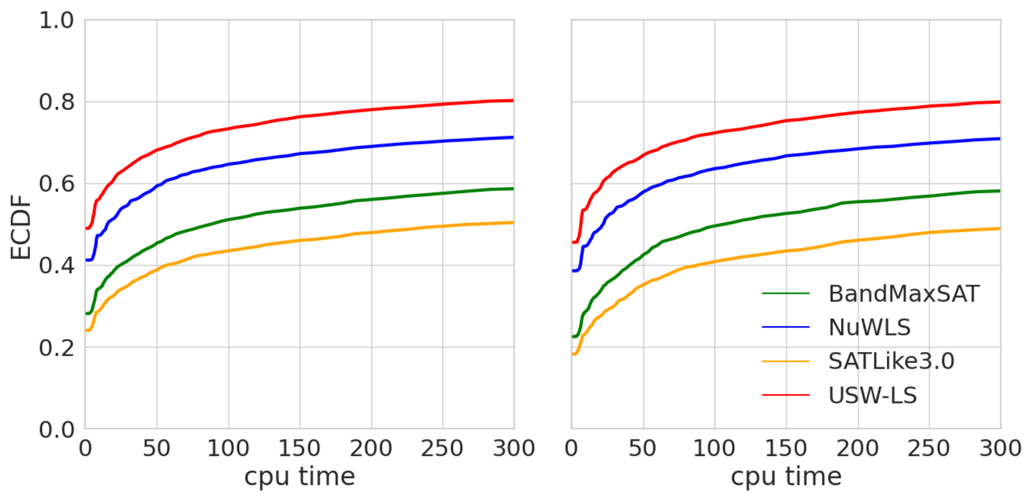


Figure 1 ECDFs of the tested solvers for two instance sets: **Left:** Unweighted, **Right:** Weighted.

5.5 Effects of uniformly adjusting soft clause weights

To examine the effect of uniformly adjusting soft clause weights, we introduce an alternative version of *USW-LS*, denoted as USW-LS-alt. The only difference between USW-LS-alt and *USW-LS* lies in their approach to adjusting soft clause weights: in USW-LS-alt, if the assignment is a feasible local optimum, only the weights of falsified soft clauses are increased by setting $k = k + 1$, and for each falsified soft clause c, $w(c) := k \times \frac{w_{ori}(c)}{avgw_{soft}}$.

Table 4 presents the results of *USW-LS* and USW-LS-alt across all the benchmarks. From table 4, *USW-LS* demonstrates superior performance over USW-LS-alt in terms of both $\#win$. and avg_{score}. On unweighted benchmarks, *USW-LS* outperforms USW-LS-alt by 45.21-71.17% in terms of $\#win$, and exhibits a 4.18-6.23% improvement over USW-LS-alt for avg_{score}. For weighted benchmarks, *USW-LS* surpasses USW-LS-alt more significantly, with 116.67-305.77% higher $\#win$ and 2.18-9.91% better avg_{score} performance.

Table 4 Comparisons of *USW-LS* with its alternative version USW-LS-alt.

Benchmark	#inst	*USW-LS*		USW-LS-alt		*USW-LS*		USW-LS-alt	
		$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}	$\#win$	avg_{score}
		60 seconds				300 seconds			
Unw_18	153	**106**	**0.7426**	73	0.6991	**113**	**0.7726**	75	0.7311
Unw_19	299	**217**	**0.7297**	142	0.6899	**226**	**0.7546**	147	0.7160
Unw_20	262	**190**	**0.7360**	111	0.6958	**197**	**0.7616**	120	0.7223
Unw_21	155	**113**	**0.6578**	77	0.6304	**119**	**0.6877**	81	0.6601
Unw_22	179	**133**	**0.7092**	83	0.6685	**140**	**0.7514**	88	0.7099
Unw_23	179	**116**	**0.6629**	75	0.6326	**121**	**0.6977**	76	0.6648
Wei_18	172	**133**	**0.7716**	58	0.7486	**134**	**0.7848**	58	0.7681
Wei_19	297	**237**	**0.7686**	72	0.7075	**244**	**0.7916**	67	0.7401
Wei_20	253	**201**	**0.7797**	57	0.7094	**211**	**0.8085**	52	0.7409
Wei_21	151	**110**	**0.6833**	44	0.6339	**114**	**0.7199**	45	0.6728
Wei_22	197	**142**	**0.7094**	43	0.6630	**152**	**0.7459**	39	0.6978
Wei_23	160	**117**	**0.6631**	54	0.6260	**119**	**0.7093**	53	0.6697

6 Conclusions and Future Work

Most solvers evaluated in the anytime tracks of recent MSEs integrate local search techniques, showing wide interest in this approach for solving MaxSAT. The main difficulty of local search is that it frequently falls into local optima, and the main technique to escape from a local optimum is clause weighting. Unfortunately, designing an effective clause weighting scheme needs to properly answer many questions, because otherwise, the solver can easily lose control of the search trajectory. In this paper, we proposed a novel clause weighting scheme called Unified-SW that provides simple, clever, and effective answers to these questions.

First, Unified-SW distinguishes between feasible local optima and infeasible local optima, to only increase the weights of hard (soft) clauses in infeasible (feasible) local optima. Second, in a feasible local optimum, instead of only increasing the weights of the falsified soft clauses as in some previous approaches, Unified-SW increases the weights of all soft clauses, whether they are satisfied or not, considering that even the satisfaction of a soft clause can represent an error that should be repaired. Third, the total increase of the weights of soft clauses is a constant in a feasible local optimum, proportionally split among all soft clauses, so that the hierarchy among the soft clauses is kept, and no upper bound is needed for their weights. Using Unified-SW, the increase of weights of soft clauses in a feasible local optimum can make the solver far from solutions, the solver then works with the new greater weights of soft clauses to hopefully move again toward a better solution via new paths, eventually by increasing the weights of hard clauses in the infeasible local optima along the paths.

We implemented Unified-SW in a new LS solver called *USW-LS*, and designed four experiments on all benchmarks from the anytime tracks of MSEs since 2018 to evaluate the effectiveness of Unified-SW, using three measures: number of winning instances, competition score used in MSEs, and anytime performance. These experiments show: (1) *USW-LS* performs much better than three state-of-the-art LS solvers NuWLS, BandMaxSAT and SATLike3.0; (2) when combined with an SAT-based solver, *USW-LS-c* also performs much better than NuWLS, BandMaxSAT and SATLike3.0 combined with the SAT-based solver; (3) Being the winning solver in all the 4 categories of the anytime tracks of MSE 2023, *USW-LS-c* also performs better than the other four top-five solvers of the anytime track of MSE 2023 on most benchmarks from the anytime tracks of MSEs since 2018; (4) Across multiple cutoff times, *USW-LS* performs also better than NuWLS, BandMaxSAT and SATLike3.0; (5) Uniformly increasing the weights of *all* soft clauses in a feasible local optimum is important, because *USW-LS* is much better than the variant that only increases the weights of falsified soft clauses.

In the future, we plan to study more deeply the landscape of the search space changed by Unified-SW for various types of instances. Currently, only falsified hard clauses see their weight increased in an infeasible local optimum, we will investigate the possibility of also increasing the weights of all soft clauses if the cost of the current assignment cannot be improved. As is said above, the total increase of weights of soft clauses in a feasible local optimum is a constant. We will investigate the possibility of increasing this constant when the search proceeds and the number of encountered feasible local optima exceeds a threshold. Finally, we also plan to apply Unified-SW to other optimization problems.

References

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

2 David Allouche, Isabelle André, Sophie Barbe, Jessica Davies, Simon de Givry, George Katsirelos, Barry O'Sullivan, Steve Prestwich, Thomas Schiex, and Seydou Traoré. Computational protein design as an optimization problem. *Artificial Intelligence*, 212:59–79, 2014.

3 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Solving (weighted) partial maxsat through satisfiability testing. In *Proceedings of SAT 2009*, pages 427–440. Springer, 2009.

4 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. A new algorithm for weighted partial maxsat. In *Proceedings of AAAI 2010*, pages 3–8, 2010.

5 Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. SAT-based MaxSAT algorithms. *Artificial Intelligence*, 196:77–105, 2013.

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

7 Gilles Audemard and Laurent Simon. On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

8 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In *Handbook of satisfiability*, pages 929–991. IOS Press, 2021.

9 Jeremias Berg, Bart Bogaerts, Jakob Nordström, Andy Oertel, and Dieter Vandesande. Certified core-guided maxsat solving. In *International Conference on Automated Deduction*, pages 1–22. Springer, 2023.

10 Jeremias Berg, Emir Demirovic, and Peter J. Stuckey. Core-boosted linear search for incomplete MaxSAT. In *Proceedings of CPAIOR 2019*, pages 39–56, 2019.

11 Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Proceedings of IJCAI 2015*, pages 747–753, 2015.

12 Shaowei Cai and Zhendong Lei. Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability. *Artificial Intelligence*, 287:103354, 2020.

13 Shaowei Cai, Chuan Luo, Jinkun Lin, and Kaile Su. New local search methods for partial MaxSAT. *Artificial Intelligence*, 240:1–18, 2016.

14 Shaowei Cai, Chuan Luo, John Thornton, and Kaile Su. Tailoring local search for partial maxsat. In *Proceedings of AAAI 2014*, pages 2623–2629, 2014.

15 Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial MAXSAT. In *Proceedings of AAAI 1997*, pages 263–268, 1997.

16 Mohamed Sami Cherif, Djamal Habet, and André Abramé. Understanding the power of max-sat resolution through up-resilience. *Artificial Intelligence*, 289:103397, 2020.

17 Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls-c-2023: Solver description. *MaxSAT Evaluation 2023*, pages 23–24, 2023.

18 Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls: Improving local search for (weighted) partial maxsat by new weighting techniques. In *Proceedings of AAAI 2023*, volume 37, pages 3915–3923, 2023.

19 Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of CP 2011*, pages 225–239, 2011.

20 Emir Demirović, Nysret Musliu, and Felix Winter. Modeling and solving staff scheduling with partial weighted maxsat. *Annals of Operations Research*, 275:79–99, 2019.

21 Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of SAT 2006*, pages 252–265, 2006.

22 Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

23 Yuejun Jiang, Henry Kautz, and Bart Selman. Solving problems with hard and soft constraints using a stochastic algorithm for max-sat. In *1st International Joint Workshop on Artificial Intelligence and Operations Research*, volume 20. Citeseer, 1995.

24 Saurabh Joshi, Prateek Kumar, Sukrut Rao, and Ruben Martins. Open-WBO-Inc: Approximation strategies for incomplete weighted MaxSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):73–97, 2019.

25 Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSAT: A partial Max-SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8(1-2):95–100, 2012.

26 Adrian Kügel. Improved exact solver for the weighted MAX-SAT problem. In *POS-10. In Proceedings of Workshop Pragmatics of SAT,, Edinburgh, UK, 2010*, volume 8, pages 15–27, 2010.

27 Javier Larrosa, Federico Heras, and Simon De Givry. A logical approach to efficient max-sat solving. *Artificial Intelligence*, 172(2-3):204–233, 2008.

28 Zhendong Lei and Shaowei Cai. Solving (weighted) partial MaxSAT by dynamic local search for SAT. In *Proceedings of IJCAI 2018*, pages 1346–1352, 2018.

29 Zhendong Lei, Shaowei Cai, Fei Geng, Dongxu Wang, Yongrong Peng, Dongdong Wan, Yiping Deng, and Pinyan Lu. Satlike-c: Solver description. *MaxSAT Evaluation*, 2021:19–20, 2021.

30 Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. In *Handbook of satisfiability*, pages 903–927. IOS Press, 2021.

31 Chu Min Li, Felip Manya, and Jordi Planes. New inference rules for max-sat. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

32 Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *Proceedings of AAAI 2010*, pages 128–133, 2010. `doi:10.1609/AAAI.V24I1.7536`.

33 Chu-Min Li, Zhenxing Xu, Jordi Coll, Felip Manyà, Djamal Habet, and Kun He. Combining clause learning and branch and bound for maxsat. In *Proceedings of CP 2021*, pages 38:1–38:18, 2021. `doi:10.4230/LIPICS.CP.2021.38`.

34 Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability. *Artificial Intelligence*, 243:26–44, 2017.

35 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In *Proceedings of CP 2014*, pages 531–548, 2014.

36 Ruben Martins, Vasco Manquinho, and Inês Lynce. Open-WBO: A modular maxsat solver. In *Proceedings of SAT 2014*, pages 438–445, 2014.

37 Alexander Nadel. Solving maxsat with bit-vector optimization. In *Proceedings of SAT 2018*, pages 54–72. Springer, 2018.

38 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In *Proceedings of FMCAD 2019*, pages 193–202, 2019.

39 Alexander Nadel. On optimizing a generic function in SAT. In *Proceedings of FMCAD 2020*, pages 205–213, 2020.

40 Alexander Nadel. Polarity and variable selection heuristics for sat-based anytime maxsat. *Journal on Satisfiability, Boolean Modeling and Computation*, 12(1):17–22, 2020.

41 Alexander Nadel. TT-Open-WBO-Inc-20: an anytime MaxSAT solver entering MSE'20. In *Proceedings of MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, pages 21–22, 2020.

42 Alexander Nadel. Introducing intel (r) sat solver. In *Proceedings of SAT 2022*, pages 8:1–8:23, 2022.

43 Alexander Nadel. Tt-open-wbo-inc-23: an anytime maxsat solver entering mse'23. *MaxSAT Evaluation 2023*, page 29, 2023.

44 Nina Narodytska and Fahiem Bacchus. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of AAAI 2014*, pages 2717–2723, 2014.

45 John Thornton, Stuart Bain, Abdul Sattar, and Duc Nghia Pham. A two level local search for MAX-SAT problems with hard and soft constraints. In *Proceedings of AI 2002*, pages 603–614, 2002.

46 John Thornton and Abdul Sattar. Dynamic constraint weighting for over-constrained problems. In *Proceedings of PRICAI 1998*, pages 377–388, 1998.

47 Furong Ye, Chuan Luo, and Shaowei Cai. Better understandings and configurations in maxsat local search solvers via anytime performance analysis. *arXiv preprint*, 2024. `arXiv:2403.06568`.

48 Jiongzhi Zheng, Kun He, Mingming Jin, Zhuo Chen, and Jinghui Xue. Combining bandmaxsat and fps with nuwls-c. *MaxSAT Evaluation 2023*, pages 25–26, 2023.

49 Jiongzhi Zheng, Kun He, and Jianrong Zhou. Farsighted Probabilistic Sampling based local search for (weighted) partial maxsat. In *Proceedings of AAAI 2023*, pages 4132–4139, 2023.

50 Jiongzhi Zheng, Kun He, Jianrong Zhou, Yan Jin, Chu-Min Li, and Felip Manya. BandMaxSAT: A local search maxsat solver with multi-armed bandit. In *Proceedings of IJCAI 2022*, pages 1901–1907, 2022.

Lazy Reimplication in Chronological Backtracking

Robin Coutelier
TU Wien, Austria

Mathias Fleury
University Freiburg, Germany

Laura Kovács
TU Wien, Austria

Abstract

Chronological backtracking is an interesting SAT solving technique within CDCL reasoning, as it backtracks less aggressively upon conflicts. However, chronological backtracking is more difficult to maintain due to its weaker SAT solving invariants. This paper introduces a lazy reimplication procedure for missed lower implications in chronological backtracking. Our method saves propagations by reimplying literals on demand, rather than eagerly. Due to its modularity, our work can be replicated in other solvers, as shown by our results in the solvers CaDiCaL and Glucose.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Automated reasoning

Keywords and phrases Chronological Backtracking, CDCL, Invariants, Watcher Lists

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.9

Supplementary Material
Software (CaDiCaL Source Code): `https://github.com/arminbiere/cadical/tree/strong-backtrack` [12]; archived at `swh:1:dir:eaf1bada31f3142996582c25a7df2118e7cacc98`
Software (NapSAT Source Code): `https://github.com/RobCoutel/NapSAT` [10]
archived at `swh:1:dir:1308f5717399bd09dcad2de805cc42eaa5504854`
Software (Glucose Source Code): `https://github.com/m-fleury/glucose` [13]
archived at `swh:1:dir:fc5f0bd80c6a9e9412c5a3f3fcde96bf17a36147`

Funding The authors acknowledge support from the ERC Consolidator Grant ARTIST 101002685; the TU Wien Doctoral College TrustACPS; the FWF SpyCoDe SFB projects F8504; the WWTF Grant ForSmart 10.47379/ICT22007; the and the Amazon Research Award 2023 QuAT; the bwHPC of the state of Baden-Württemberg; the German Research Foundation (DFG) through grant INST 35/1597-1 FUGG; and a gift from Intel Corporation.

Acknowledgements The first author's Master thesis [11] conducted at the University of Liège under the supervision of Pascal Fontaine laid the foundations of this work. We thank him for his insights.

1 Introduction

In the past few years, chronological backtracking in CDCL-based SAT solving attracted renewed interest as it implements less aggressive procedures when backtracking upon conflicts, particularly for undoing literal assignments stored in the assignment stack. Chronological backtracking has been proven sound and complete, while also empirically improving performance on SAT competition problems [18, 20, 21].

Without chronological backtracking in SAT solving, the truth value of each literal is set as early as possible in the solving process. With chronological backtracking, there are, however, missed lower implications (MLI), i.e., clauses that could have set a literal at a lower SAT decision level. As a remedy to MLI, IntelSAT [20] and CaDiCaL-1.9.4 [22] fix the level

Invariant properties for CDCL algorithms (Section 3)

Invariant 1–*Weak watched literals*:	No conflict is missed.

Invariants on implications, native for NCB and WCB (Section 3.1)

Invariant 2–*Implied literals*:	Literals are decisions or implied by a clause C that is made unit by the partial assignment.
Invariant 3–*Topological order*:	The partial SAT assignment follows a topological order of the implication graph.

Strong invariant, non-trivial for CDCL with CB and native in NCB (Section 3.1)

Invariant 4–*Strong watched literals*:	No implication nor conflict can be missed.

Figure 1 Invariant properties for CDCL-based SAT solving and maintained by the different chronological backtracking (CB) strategies, particularly by non-chronological backtracking (NCB) and weak chronological backtracking (WCB).

of the assignments. Modifying levels impacts solving performance and significantly clutters the code; for example, reimplication techniques for detecting MLI have been removed in CaDiCaL-1.9.5 [4] due to the increased code complexity.

In this paper, we introduce a *lazy reimplication procedure for resolving missed lower implications in chronological backtracking, while also ensuring efficiency in SAT solving.* Doing so, in Figure 1, we state the invariant properties to be maintained during CDCL and highlight differences between relevant backtracking approaches in SAT solving. In particular, we consider and adjust variants of *non-chronological backtracking* (NCB) [23] and strong chronological backtracking (SCB) [20]. A formal presentation of these invariants and backtracking variants is given in Section 3. Using the invariants of Figure 1, in Section 4 we introduce a lazy reimplication procedure to handle missed lower implications, with a particular focus on handling unit implications after backtracking. We also adjust and enhance the first unique implication point (UIP) algorithm [19] with the knowledge of missed lower implications. Our approach is sound (Section 5). We implemented our work in the new solver NapSAT [10] and present our empirical findings in Section 6. To demonstrate the flexibility of our lazy reimplication techniques, we also implemented the algorithms of Section 4 in CaDiCaL [5] and Glucose [1], and provide empirical comparisons using these solvers.

Related work. Within CDCL, the truth values of literals are assigned by guessing (deciding) and propagating them in a trail until a conflict is found. Upon conflict analysis, the trail is adapted by backtracking, i.e. revoking some assignments and swapping the truth value of one variable, called the unique implication point (UIP). The standard approach [23] is to fix the conflict as early as possible with *non-chronological backtracking* (NCB) and all assignments between the current point and the point where the UIP is set are deleted.

A different backtracking approach comes with *chronological backtracking* (CB) [18, 21]. Here, a less aggressive backtracking scheme is used and some propagations and decisions are kept. Chronological backtracking may backjump at any level between the UIP and the UIP falsification point minus one. As a result, chronological backtracking resets a smaller part of the trail, but it may miss propagations that could have been done earlier if the learned clause was known beforehand. In this paper, we refer by *weak chronological backtracking* (WCB) to the CDCL algorithms that use chronological backtracking mechanism and which do not detect every propagation as early as possible (see Section 3.2)

For recovering such missed propagations, we define *strong chronological backtracking* (SCB). In particular, Nadel [20] introduced a reimplication procedure that eagerly re-assigns literals detected as missed lower implications to their lowest possible level. We refer to this SCB technique as *eager strong chronological backtracking* (ESCB). Our work introduces a new SCB method, *lazy strong chronological backtracking* (LSCB). Unlike ESCB, within LSCB we reimply missed implications on demand. As such, our work is stronger than WCB, as WCB does not perform reimplications at all. In addition, our technique is shown to be easier and more flexible to implement than ESCB or WCB (Section 6).

Our contributions. This paper brings the following contributions to chronological backtracking in CDCL-based proof search.

1. We formalize invariant properties that need to be maintained during SAT solving with chronological backtracking (Section 3). Our invariants incorporate and reason over different backtracking strategies.
2. We introduce *lazy strong chronological backtracking* (LSCB) for on-demand reimplication of (conflict) (Sec. 4) and prove soundness of our approach (Section 5).
3. We implement our work in the new NapSAT [10, 14] solver (Section 6). To showcase the flexibility and efficiency of our approach, we integrate LSCB into CaDiCaL [5] and Glucose [1], and provide experimental comparisons using these solvers.

2 Preliminaries

We assume familiarity with propositional logic and CDCL [6], and use the standard logical connectives $\neg$, $\wedge$, and $\vee$. A finite set of elements (e.g. literals) is called *conjunctive* (respectively, *disjunctive*) to indicate that the set is the conjunction (respectively, disjunction) of its elements. An *ordered set* is a set $\mathcal{S}$ which defines a bijective function $p_{\mathcal{S}}$ from elements of $\mathcal{S}$ to naturals, such that $p_{\mathcal{S}}(e)$ is the position of the element e in the ordered set $\mathcal{S}$. We consider the first element of $\mathcal{S}$ to have the position 0. Ordered sets are stable under the removal of elements; that is, for the ordered sets $\mathcal{S}, \mathcal{T}, \mathcal{U}$ with $\mathcal{S} = \mathcal{T} \setminus \mathcal{U}$, we have $\forall e, e' \in \mathcal{S}.\ p_{\mathcal{S}}(e) < p_{\mathcal{S}}(e') \Leftrightarrow p_{\mathcal{T}}(e) < p_{\mathcal{T}}(e')$. We denote by $\cdot$ set concatenation; for simplicity, we use $\cdot$ to also denote appending a sequence with an element. We write $\mathcal{S}[a : b]$ to select the ordered elements e in $\mathcal{S}$ with positions $a \leq p_{\mathcal{S}}(e) \leq b$.

We denote by $\mathcal{V}$ a countable set of Boolean variables v. We consider propositional formulas F in conjunctive normal form (CNF), represented by a conjunctive set of clauses $\{C_1, C_2, \ldots, C_n\}$ over $\mathcal{V}$. Clauses are disjunctive sets of literals $C = \{c_1, c_2, \ldots, c_m\}$, where a literal c_i is either a Boolean variable v or a negation $\neg v$ of a variable v.

To efficiently identify unit propagations, SAT solvers track two literals per clause in the two-watched literal scheme [19]. We denote the watched literals of a clause C by c_1 and c_2, and write $\mathrm{WL}(c_1)$ and $\mathrm{WL}(c_2)$ for the watched lists of c_1 and c_2. We have $C \in \mathrm{WL}(c_1) \cap \mathrm{WL}(c_2)$.

During SAT solving, solvers keep track of a *partial assignment*, also called *trail* and denoted as the conjunctive ordered set $\pi = \tau \cdot \omega$, which is split into two parts: (i) τ is the set of literals that were already propagated and do not need to be inspected anymore (by checking the watch lists); (ii) ω is the *propagation queue* containing literals that were implied and waiting to be propagated. The partial assignment π contains the set $\pi^d \subseteq \pi$ of decision literals. Decisions literals in π^d are arbitrarily chosen literals when unit propagation cannot be further used and the truth value of a (decision) literal needs to be picked and assigned. We call *unit*, a clause C containing exactly one unassigned literal ℓ and whose other literals are falsified, i.e., $\exists \ell \in C.\ C \setminus \{\ell\}, \pi \models \bot \wedge |\ell| \notin |\pi|$. For conflict analysis, the propagation reasons of literals are analyzed. Therefore, SAT solvers use a ρ function that maps literals to clauses such that $\rho(\ell)$ captures the reason for propagating ℓ. The reason for propagating ℓ is the clause C that implied ℓ under assumption π, that is, $[\ell \in \pi] \wedge [\ell \in \rho(\ell)] \wedge [\rho(\ell) \setminus \{\ell\} \wedge \pi \models \bot]$. Following [18], we use δ to represent the (decision) *level of* ℓ, i.e., the level when a truth assignment to ℓ was made. Formally, if ℓ is a decision literal, then the level $\delta(\ell) = \delta(\neg\ell)$ of ℓ is the number of decisions preceding and including ℓ, that is $\delta(\ell) = |\pi[0 : p_\pi(\ell)] \cap \pi^d|$. Further, for literals ℓ implied by $\rho(\ell)$, we have $\delta(\ell) = \max_{\ell' \in \rho(\ell) \setminus \{\ell\}} \delta(\ell')$. Finally, $\delta(\ell) = \infty$ for unassigned literals ℓ. The definition of δ is extended to clauses and trails, with $\delta(C) = \max_{\ell \in C} \delta(\ell)$; similarly for $\delta(\pi)$. The level of the empty set is $\delta(\emptyset) = 0$. We write $\delta[\ell \leftarrow d]$ to denote that the level of ℓ is updated to d. We reserve the special symbol $\blacksquare$ to denote *undefined clauses* during SAT solving, with $\delta(\blacksquare) = \infty$.

In standard CDCL with non-chronological backtracking (NCB) [23], level δ stores the number of decisions that appear before in the trail, and is always the lowest level possible. In CDCL with chronological backtracking, the history of propagations and conflicts may, however, lead to *missed lower implications* (MLI), where a MLI captures the fact that a clause C is satisfied by a unique literal ℓ at a level *strictly higher* than $\delta(C \setminus \{\ell\})$. Therefore, in a MLI, the literal ℓ could have been propagated at a lower level in the trail.

▶ **Example 1** (Missed Lower Implications – MLI)**.** Figure 2 shows a clause set $\{C_1, \dots, C_7\}$ and a trail $\pi = \tau \cdot \omega$ during CDCL solving with chronological backtracking. The trail diagram displays, from left to right, the order in which literals are decided and propagated, as well as the location of the propagation head (symbolized by the dashed line). The propagation level is symbolized by the height of the step. As a visual aid, literals are colored green, red or black, symbolizing respectively satisfied, falsified, and unassigned literals. The watched literals are the first two in the clause.

In the example, the clause set $F_0 = \{C_1, \dots, C_6\}$ was given as input. The decisions v_1, v_2 and v_3 were made, then the solver found a conflict in C_2 after implying v_4 with reason $\rho(v_4) = C_1$. The clause $C_7 = \neg v_3@3 \vee \neg v_1@1$ is learned and the solver backtracks to level $\delta(C_7) - 1 = 2$, continuing its propagations until it reaches the assignment shown on Figure 2. Figure 2 shows that C_4 is a MLI. Indeed, v_2 is satisfied at level 2, while all other literals are falsified at level 1. After backtracking to level 1, the implication of v_2 by C_4 is missed since (i) $\neg v_3$ was already propagated, and (ii) C_4 is watched by v_3 and v_2.

3 Invariant Properties on CDCL Variants

To properly handle MLI similar to Example 1, in this section we revisit and formalize our invariants from Figure 1, expressing properties that need to be maintained in (variants of) CDCL with chronological backtracking.

$$\begin{aligned}
C_1 &= \neg v_3@1 \lor \quad v_4 \\
C_2 &= \neg v_3@1 \lor \neg v_4 \quad \lor \neg v_1@1 \\
C_3 &= \quad v_5@1 \lor \quad v_3@1 \\
C_4 &= \quad v_2@2 \lor \quad v_3@1 \lor \neg v_5@1 \\
C_5 &= \quad v_6@2 \lor \neg v_5@1 \lor \quad v_3@1 \\
C_6 &= \neg v_6@2 \lor \neg v_2@2 \lor \neg v_5@2 \\
C_7 &= \neg v_3@1 \lor \neg v_1@1
\end{aligned}$$

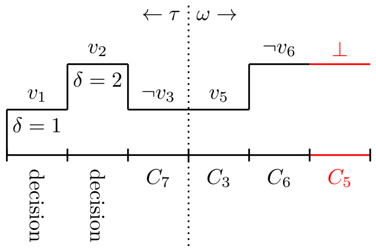


Figure 2 C_4 is a MLI while C_5 would be a MLI if v_5 was set to true. We use the notation $v@1$ to indicate that literal v is on level 1.

The crux of our invariant properties is captured by watched literals [19]. They reduce the number of clauses to be checked when propagating a literal. Invariant 1 therefore expresses that, as long as CDCL does not falsify one of the watched literals c_1, c_2 of a clause C, the clause C is not a conflict. Therefore, when propagating a watched literal c_i during CDCL, only checking the clauses watched by $\neg c_i$ is sufficient to not miss any conflict.

▶ **Invariant 1** (Weak watched literals). *Let $\pi = \tau \cdot \omega$ be the current trail. For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow \neg c_2 \notin \tau$.*

Invariant 1 ensures that conflicts are not missed during CDCL. Indeed, if there is a conflicting clause C, the conflict is found after propagating all literals of C. After propagation, no more literal has to be propagated, so $\pi = \tau$. A conflicting clause C thus violates Invariant 1, and hence the conflict of C is captured during CDCL.

3.1 CDCL Invariants on Implications

We next ensure the soundness of unit implications. Invariant 2 expresses that literals are either decisions or implied by a sound implication. Note that an implication can be performed if there is only one unassigned literal that can satisfy a clause C; hence, C is a unit clause. In addition to ensuring that the solver infers correct literals, Invariant 2 is also relevant for conflict analysis (see proof of Theorem 13).

▶ **Invariant 2** (Implied literals). *If a literal ℓ is in the trail π, then ℓ is either a decision literal or ℓ is implied by π and its reason $\rho(\ell)$. That is,*

$$\forall \ell \in \pi.\ \ell \in \pi^d \ \lor\ [\ell \in \rho(\ell) \ \land\ [\rho(\ell) \setminus \{\ell\} \land \pi] \vDash \bot].$$

To perform conflict analysis with the first unique implication point (UIP) [19], CDCL solving assumes that literals are organized in a topological sort of the implication graph.

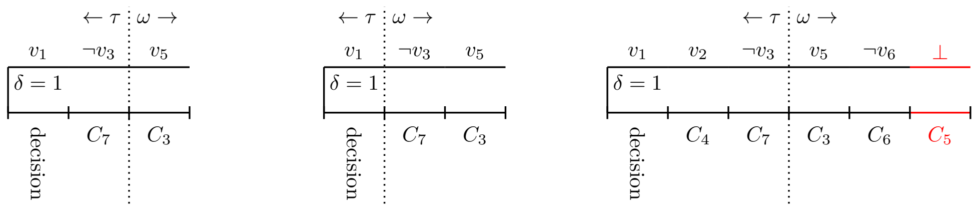
▶ **Invariant 3** (Topological order). *Trail π is a topological order of the implication graph:*

$$\forall \ell \in \pi.\ \forall \ell' \in \rho(\ell).\ p_\pi(\neg \ell') \le p_\pi(\ell),$$

where $p_\pi(\ell)$ and $p_\pi(\neg \ell')$ are respectively the positions of ℓ and $\neg \ell'$ in π.

Invariant 3 holds by construction in CDCL with non-chronological backtracking (NCB) and chronological backtracking without reimplication. However, Invariant 3 is crucial in any setting of reimplying literals.

Finally, we impose that Boolean constraint propagation (BCP) in CDCL does not miss unit implication during proof search. Invariant 4 therefore formalizes that CDCL cannot have one propagated falsified watched literal without the clause being satisfied.



(a) Weak CB. **(b)** Restoration SCB. **(c)** Eager SCB.

Figure 3 Different CB ways of handling the missed lower implications of Figure 2.

▶ **Invariant 4** (Strong watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow c_2 \in \pi$.*

Invariant 4 strengthens Invariant 1. When a conflicting clause C is detected while propagating ℓ, the literal ℓ cannot be added to τ without violating Invariant 4. As such, by imposing Invariant 4, the conflict of C is resolved and the trail is adapted.

3.2 Chronological Backtracking

Invariant 4 holds for CDCL with NCB, since the trail contains monotonically increasing decision levels. Therefore, within NCB, literals are unassigned in the reverse order of propagation. In particular, if a literal ℓ is satisfied in a clause C when propagating another literal ℓ', the literal ℓ remains satisfied at least until ℓ' is backtracked.

Weak chronological backtracking (WCB). When considering (variants of) chronological backtracking in CDCL, Invariant 4 becomes critical, as detailed next. The core idea is to save parts of the trail without repropagating unlike [15].

▶ **Example 5.** Let us revisit the example of Figure 2. Figure 3a shows the trail after backtracking to level 1. Literal v_3 is already propagated ($v_3 \in \tau$), and C_4 is still watched by v_2 and v_3. Therefore, the implication of v_2 is missed, even though Invariant 1 is not violated.

To circumvent the problem of missing implications similar to Example 5, we distinguish a *weak chronological backtracking (WCB)* variant of CDCL with chronological backtracking. Within WCB, Invariant 4 is not necessarily satisfied, as unit implications at lower levels can be missed. To recover Invariant 4 in variants of CDCL with CB, we adjust and label two existing solutions in SAT solving: (i) *restoration* [21], for repairing the trail p after backtracking; and (ii) *prophylaxis* [20], for forcing literals at the lowest possible level.

Restoration. We call *restoration* the approach in which the trail π is repaired by pushing back the propagation head when propagating [21]. Out-of-order literals are repropagated whenever they are moved in the trail during backtracking. For example, in Figure 3b, v_3 was the first literal that changed position during backtracking, so this is where the propagation head is set. When backtracking to level δ, the propagation head is set to $p_\pi(\pi^d[\delta])$. When v_3 is repropagated, v_2 is reimplied. We, therefore, restore Invariant 4 by repropagating the out-of-order literals. We call this approach *restoring strong chronological backtracking* (RSCB), allowing to restore the trail π by propagating more. It is also used in CaDiCaL [5].

Table 1 CB variants in CDCL, together with their invariant properties.

	Inv. 1	Inv. 4	Inv. 6	Inv. 9	Solvers
NCB	✓	✓	✓	✓	Most CDCL solvers
WCB	✓	✗	✗	✗	Our work – NapSAT
RSCB	✓	✓	✗	✗	Maple_LCM_Dist [21], CaDiCaL
ESCB	✓	✓	✓	✓	IntelSAT and CaDiCaL 1.9.4
LSCB	✓	✓	✗	✓	Our work – NapSAT, now in CaDiCaL

Prophylaxis. We name *prophylaxis*[1] the approach in which missed lower implications are prevented from becoming missed unit implications [20]. Prophylaxis uses an eager reimplication procedure and imposes the validity of a compatibility invariant; we formalize this property in Invariant 6. That is, when a clause C is detected to be a missed lower implication of ℓ, then ℓ is reimplied at level $\delta(C \setminus \{\ell\})$ and its reason for propagation is updated. Prophylaxis thus enforces our *backtrack compatible* Invariant 6 by ensuring that no clause can become unit after backtracking. Furthermore, Invariant 6 guarantees that literals are always propagated at the lowest level, and conflicts are detected at the lowest level.

▶ **Invariant 6** (Backward compatible watched literals). *For each clause $C \in F$ watched by the two distinct watched literals c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow [c_2 \in \pi \wedge \delta(c_2) \leq \delta(c_1)]$.*

▶ **Example 7.** Figure 3c shows the trail after v_2 is reimplied. In this case, v_2 was a decision, and $\neg v_6$ has to be reimplied to level 1 as well. All literals are propagated at the lowest possible level. Thus, using Invariant 6, the conflict C_5 is properly detected at level 1, instead of level 2. Figure 3c also shows that the trail π no longer follows a topological order of the implication graph. These issues have to be addressed.

Based on Invariant 6, *eager strong chronological backtracking (ESCB)* is used in [20, 22], yielding a CDCL method with chronological backtracking that satisfies Invariant 6 by eager reimplication of missed lower implications. In Table 1 we summarize backtracking strategies in CDCL, also listing our solution in this respect: *lazy reimplication in strong chronological backtracking (LSCB).* Our LSCB approach maintains Invariant 1 and Invariant 4, while weakening Invariant 6 via Invariant 9, as described next in Section 4 and implemented in Algorithm 1.

4 Adapting CDCL with Lazy Reimplications

Embedding the prophylaxis approach of Section 3.2 in existing CDCL data structures is highly non-trivial, due to the rigid and entangled data structures [20, 22], see e.g. [11]. In addition, reimplying literals [20, 22] changes the implication graph, and hence the trail π is no longer a topological sort of the implications; as such, Invariant 3 must be restored.

While the restoration approach of Section 3.2 offers a practically simpler solution, restoration might require the re-propagation of a large part of π and thus can be computationally very expensive. For example, while in Figure 3b only one literal had to be re-propagated, re-propagation could be applied on an arbitrary number of literals.

[1] "Prophylaxis" is a chess term referring to a move that deals with a threat before it becomes a problem.

Our solution: Lazy reimplication in CDCL. To overcome inefficiencies of restoration and pure prophylaxis, our work advocates a *lazy reimplication* technique for CDCL with strong chronological backtracking. To ensure Invariant 4, we reimply literals after backtracking. That is, *we detect missed lower implications eagerly but reimply them lazily.*

Our lazy reimplication approach for CDCL-based solving is summarized in Algorithm 1. In what follows, we describe the key ingredients of Algorithm 1 and revise the CDCL invariants of Section 3, adjusted to Algorithm 1. To this end, we introduce a lazy reimplication vector λ to store missed lower implications, where λ is a function from literals to clauses. Intuitively, the lazy reimplication vector λ stores the lowest

detected missed lower implication for each literal ℓ. The clause $\lambda(\ell) \neq \blacksquare$ is an alternative reason that would propagate ℓ in trail π, lower than the reason $\rho(\ell)$. Initially, no clause is assigned, and $\forall\ell.\ \lambda(\ell) = \blacksquare$ (that is, the undefined clause). Invariant 8 is asserted to hold during proof search.

▶ **Invariant 8** (Lazy reimplication). *If the lazy reimplication reason $\lambda(\ell)$ of literal ℓ is defined, then the clause $\lambda(\ell)$ is a missed lower implication of ℓ. That is,*

$$\begin{aligned}\lambda(\ell) \neq \blacksquare \ \Rightarrow \quad & \ell \in \pi \wedge\ \ell \in \lambda(\ell) \\ & \wedge \big(\lambda(\ell) \setminus \{\ell\} \wedge \pi\big) \vDash \bot \\ & \wedge\ \delta(\lambda(\ell) \setminus \{\ell\}) < \delta(\ell)\end{aligned}$$

When a missed lower implication for ℓ is detected, then ℓ is not reimplied directly. Rather, we store the MLI in λ until ℓ is unassigned during backtracking. For example, if a literal ℓ is assigned at level 3 and a missed lower implication C for ℓ is detected with $\delta(C \setminus \{\ell\}) = 1$, then backtracking to level 2 will reassign ℓ from level 3 to level 1 by C.

Using our lazy reimplication vector λ, we weaken Invariant 6 into Invariant 9 such that, during backtracking, we identify missed lower implications without requiring the re-propagation of out-of-order literals.

▶ **Invariant 9** (Lazy backtrack compatible watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$, if one watched literal c_1 of C is falsified by τ, then the other c_2 must be satisfied at a lower level, or a missed lower implication lower than c_1 is set in λ.*

$$\neg c_1 \in \tau \Rightarrow \Big(c_2 \in \pi \wedge \big(\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)\big)\Big)$$

Lazy reimplication for strong chronological backtracking – LSCB. Guided by the reimplication and backtracking properties of Invariant 8 and Invariant 9, Algorithm 1 shows our LSCB algorithm for CDCL with chronological backtracking, as a slight refactoring of weak chronological backtracking (WCB). In the following algorithms, particularities of LSCB are highlighted in blue.

An important detail should be noted upon Algorithm 1: in our abstract representation, it is not explicitly checked whether the learned clause D is different from the conflicting clause C; such a check, however, should be performed when implementing Algorithm 1. Indeed, as pointed out in RSCB [18], it is possible that a conflicting clause C does not require conflict analysis since C might already be a UIP. However, if the highest literal ℓ in C is a MLI, then the clause might be conflicting again after backtracking (see Algorithm 4).

▶ **Example 10.** Consider the example of Figure 2. Here, the conflicting clause C_5 only has one literal at the highest level, and, as such, it qualifies as a UIP. Therefore no conflict analysis is required, we only backtrack to level 1, and then C_5 implies v_6 at level 1. However, if $\neg v_6$ was a missed lower implication, then backtracking to level 1 would reimply $\neg v_6$, with C_5 conflicting again; this time, however, C_5 would require conflict analysis.

Algorithm 1 Lazy Reimplication in CDCL with CB.

```
1: $\pi = \tau = \omega = \pi^d = \emptyset$
2: $\forall \ell.\ \delta(\ell) = \infty$
3: $\forall \ell.\ \mathrm{WL}(\ell) = \emptyset$
4: $\forall \ell.\ \rho(\ell) = \lambda(\ell) = \blacksquare$
5: procedure CDCL($F$)
6:     for $C \in F$ do                          ▷ Fill the watcher lists
7:         $c_1, c_2 \leftarrow$ two literals in $C$
8:         $\mathrm{WL} \leftarrow \mathrm{WL}[c_1 \leftarrow \mathrm{WL}(c_1) \cup \{C\}][c_2 \leftarrow \mathrm{WL}(c_2) \cup \{C\}]$
9:     while $\top$ do
10:        $C \leftarrow \mathrm{BCP}()$                        ▷ Algorithm 2
11:        if $C = \top$ then
12:            if $|\pi| = |\mathcal{V}|$ then          ▷ All variables are assigned
13:                return SAT
14:            $\ell \leftarrow$ DECIDE()
15:            $\omega \leftarrow \omega \cdot \ell, \pi^d \leftarrow \pi^d \cdot \ell, \delta \leftarrow \delta[\ell \leftarrow |\pi^d|]$
16:            continue
17:        $D \leftarrow$ ANALYZE($C$)                   ▷ Algorithm 4
18:        if $\delta(D) = 0$ then
19:            return UNSAT
20:        $d \leftarrow$ any level between $\delta(D) - 1$ and the second highest level of $D$
21:        BACKTRACK($d$)                       ▷ Algorithm 3
22:        $\ell \leftarrow$ the unassigned literals in $D$
23:        $c_2 \leftarrow$ the second highest literal in $D$
24:        $\omega \leftarrow \omega \cdot \ell, \delta \leftarrow \delta[\ell \leftarrow \delta(C \setminus \{\ell\}], \rho \leftarrow \rho[\ell \leftarrow D]$
25:        $F \leftarrow F \cup \{D\}$                       ▷ Does nothing if $C = D$
26:        $\mathrm{WL} \leftarrow \mathrm{WL}[\ell \leftarrow \mathrm{WL}(\ell) \cup \{D\}][c_2 \leftarrow \mathrm{WL}(c_2) \cup \{D\}]$
```

Propagation in LSCB. When falsifying a watched literal, Algorithm 1 might need to find a replacement candidate to become the new watched literal (line 7 of Algorithm 2). We define the property of the candidate literal with information about its level as below.

▶ **Definition 11** (Candidate literal). *Let clause C be watched by the literals c_1 and c_2. with $\neg c_1 \in \omega$. Then,* SEARCHREPLACEMENT*(C, c_1, c_2) from Algorithm 2 returns a* candidate literal *r for which one of the following holds:*

- *Invariant 6 is satisfied on C after $\neg c_1$ is added to τ, i.e.*
  $\neg r \in (\tau \cdot \neg c_1) \Rightarrow c_2 \in \pi \wedge \delta(c_2) \leq \delta(r)$;
- *C is conflicting, propagating, or a MLI for c_2. As such, $C \setminus \{c_2\}$ is unsatisfiable with the current assignment, and r is at the highest decision level in $C \setminus \{c_2\}$, that is*
  $\big(C \setminus \{c_2\} \wedge \pi\big) \vDash \bot \quad \wedge \quad \delta(r) = \delta(C \setminus \{c_2\})$

Concretely, the SEARCHREPLACEMENT(C, c_1, c_2) procedure iterates over literals of $C \setminus \{c_2\}$ and stops when it finds a literal r that would satisfy Invariant 6 if c_1 was replaced by r. In case of failure, it returns the highest literal in $C \setminus \{c_2\}$. The knowledge of the highest literal in $C \setminus \{c_2\}$ is enough to determine the nature and level of the clause.

Algorithm 2 shows our Boolean constraint propagation (BCP) algorithm adapted to support LSCB. As opposed to standard BCP, Algorithm 2 does not stop when the other watched literal is satisfied. We need the extra guarantee that either c_2 is implied at a level

Algorithm 2 Boolean Constraint Propagation in LSCB.

```
1: procedure PropagateLiteral($\ell$)
2:     $c_1 \leftarrow \neg\ell$
3:     for $C \in \mathrm{WL}[c_1]$ do
4:         $c_2 \leftarrow$ the other watched literal in $C$
5:         if $c_2 \in \pi \wedge$ $[\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)]$ then
6:             continue
7:         $r \leftarrow$ SearchReplacement($C, c_1, c_2$)
8:         $\mathrm{WL} \leftarrow \mathrm{WL}[c_1 \leftarrow \mathrm{WL}(c_1) \setminus \{C\}][r \leftarrow \mathrm{WL}(r) \cup \{C\}]$
9:         if $\neg r \notin \pi$ then
10:            continue
11:        if $\neg c_2 \in \pi$ then                          ▷ Conflict
12:            return $C$
13:        if $c_2 \in \pi$ then
14:            if $\delta(c_2) > \delta(r) \wedge \delta(\lambda(c_2) \setminus \{c_2\}) > \delta(r)$ then
15:                $\lambda \leftarrow \lambda[c_2 \leftarrow C]$         ▷ New or improved MLI
16:            continue
17:        $\omega \leftarrow \omega \cdot c_2, \rho \leftarrow \rho[c_2 \leftarrow C], \delta \leftarrow \delta[c_2 \leftarrow \delta(r)]$
18:    return $\top$
1: procedure BCP
2:     while $\omega \neq \emptyset$ do
3:         $\ell \leftarrow$ First($\omega$)
4:         $C \leftarrow$ PropagateLiteral($\ell$)
5:         if $C \neq \top$ then
6:             return $C$
7:         $\omega \leftarrow \omega \setminus \{\ell\}, \tau \leftarrow \tau \cdot \ell$
8:     return $\top$
```

lower than c_1, or it is registered as a MLI before skipping the clause. Further, when a non-falsified replacement literal cannot be found, Algorithm 2 still changes the watched literal. While this is not always strictly necessary (for example, in conflicts), systematically swapping the highest literal allows checking the level of the clause in constant time and provides cheap useful properties to the clause.

Backtracking in LSCB. When backtracking, our LSCB approach has the information of whether a clause C violates Invariant 4. Therefore, Algorithm 3 can directly imply those missed lower implications (line 15 of Algorithm 3).

The order in which literals are reimplied in Algorithm 3 is not important, as shown later in Theorem 17. It is, however, unclear whether a specific order would impact performance in problems where the stability of literal position in the trail is important. In such cases, ordering the reimplications in increasing levels might be beneficial.

Conflict analysis with MLI. As opposed to traditional backtracking, Algorithm 1 does not guarantee that, once it backtracks to a level lower than the level of the learned clause D, the clause D will be propagating. Indeed, let the falsified learned clause $D = \{c_1, c_2, \ldots, c_m\}$ with c_1 a unique literal at level $\delta(D)$. If we backtrack to level $\delta(D) - 1$, c_1 might be reimplied at a lower level, and D would still be a conflict. In response to this, we propose the following two solutions:

Algorithm 3 Backtracking and Reimplication.

```
1: procedure BACKTRACK($d$)
2:     $\Lambda \leftarrow \emptyset$                                   ▷ $\Lambda$ is the set that will be reimplied
3:     $\pi = \tau \cdot \omega$
4:     for $\ell \in \pi$ do
5:         if $\delta(\ell) > d$ then
6:             if $\delta(\lambda(\ell) \setminus \{\ell\}) \leq d$ then
7:                 $\Lambda \leftarrow \Lambda \cup \{\lambda(\ell)\}$           ▷ Store the MLI for later
8:             $\pi \leftarrow \pi \setminus \{\ell\}$                     ▷ Unassign $\ell$
9:             $\delta \leftarrow \delta[\ell \leftarrow \infty], \rho \leftarrow \rho[\ell \leftarrow \blacksquare]$
10:            $\lambda \leftarrow \lambda[\ell \leftarrow \blacksquare]$                 ▷ $\lambda(\ell)$ is either used, or no longer valid
11:    $\pi^d \leftarrow \pi \cap \pi^d$                                ▷ Remove the unassigned literals
12:    $\tau \leftarrow \pi \cap \tau$
13:    $\omega \leftarrow \pi \setminus \tau$
14:    for $C \in \Lambda$ do                               ▷ Reimplying the MLI
15:        $\ell \leftarrow$ the unassigned literal in $C$
16:        $\omega \leftarrow \omega \cdot \ell, \rho \leftarrow \rho[\ell \leftarrow C], \delta \leftarrow \delta[\ell \leftarrow \delta(C \setminus \{\ell\})]$
```

(Analyze-1) we analyze the conflict and backtrack again until we get a unit clause;

(Analyze-2) we perform conflict analysis with the knowledge of missed lower implications.

In Algorithm 4 we chose option 2. Option 1 will generate the same clause in the end, but might create some unnecessary ones in the process. We empirically check our intuition in Section 6 and demonstrate that option 2 indeed works better. We refer to $D \otimes_{\ell} C'$ as the result of binary resolution applied to the clauses C and D over the literal ℓ.

In Algorithm 4, when possible, we use the lazy reimplication reason $\lambda(\ell)$ instead of the real reason $\rho(\ell)$ during conflict analysis. The lazy reason $\lambda(\ell)$ is guaranteed to introduce literals at a level lower than $\delta(C)$, making it converge to a UIP faster. Once a UIP is obtained, Algorithm 4 does not stop if there exists a missed lower implication for the last literal at the conflict level. Furthermore, we adapted the learnt clause minimization approach [24], adjusted to Algorithm 4 so that both reasons are checked if the literal can be removed.

▶ **Example 12.** Figure 4 shows a conflict after Algorithm 2 detected a missed lower implication C_6. From Invariant 9, we have $\lambda(\neg v_3) = C_6$. Algorithm 1 will then trigger Algorithm 4 to analyse the conflict on C_5. During conflict analysis with Algorithm 4, we start from the conflicting clause $D = \neg v_7@2 \vee v_5@2 \vee v_6@1$ and apply the resolution $D \leftarrow D \otimes_{\neg v_7} C_4$ to obtain $D = v_5@2 \vee v_3@2 \vee v_6@1$. We once again apply resolution and have $D \leftarrow D \otimes_{v_5} C_2$, yielding the clause $D = v_3@2 \vee v_6@1 \vee \neg v_4@1$. As this D is a UIP, most CDCL approaches would stop conflict analysis here. However, in our LSCB approach we know that v_3 can be reimplied at level 1. Therefore, after backtracking to level 1 and reimplying $\neg v_3$ with Algorithm 3, the clause D would still be conflicting and conflict analysis would need to be triggered again. Instead we apply the resolution $D \leftarrow D \otimes_{v_3} C_6$ to get a clause at level 1, namely clause $D = v_6@1 \vee \neg v_4@1 \vee v_2@1$. We then continue until the procedure at level 1 and obtain the final clause $D = v_2@1$.

Algorithm 4 Conflict Analysis.

```
procedure ANALYZE($C$)
    $\pi \leftarrow \tau \cdot \omega$                                  ▷ Array version of the trail.
    $D \leftarrow C$                                        ▷ Current learned clause.
    $n \leftarrow |\{\ell : \ell \in D \wedge \delta(\ell) = \delta(D)\}|$        ▷ Number of literals at the highest level.
    while $\top$ do
        $\ell \leftarrow$ the last literal in $\pi$ falsified in $D$ at level $\delta(D)$
        if $n = 1 \wedge \lambda(\ell) = ■$ then
            return $D$
        $C' \leftarrow \rho(\ell)$
        if $\lambda(\ell) \neq ■$ then
            $C' \leftarrow \lambda(\ell)$
        $D \leftarrow D \otimes_\ell C'$
        $n \leftarrow |\{\ell : \ell \in D \wedge \delta(\ell) = \delta(D)\}|$
```

$$\begin{aligned}
C_1 &= \neg v_2@1 \vee v_1@1 \\
C_2 &= \neg v_5@2 \vee v_3@2 \vee \neg v_4@1 \\
C_3 &= \neg v_6@1 \vee v_2@1 \vee \neg v_4@1 \\
C_4 &= v_7@2 \vee v_5@2 \vee v_3@1 \\
C_5 &= v_5@2 \vee \neg v_7@2 \vee v_6@1 \\
C_6 &= \neg v_3@2 \vee \neg v_4@1 \vee v_2@1 \\
C_7 &= v_4@1 \vee v_2@1
\end{aligned}$$

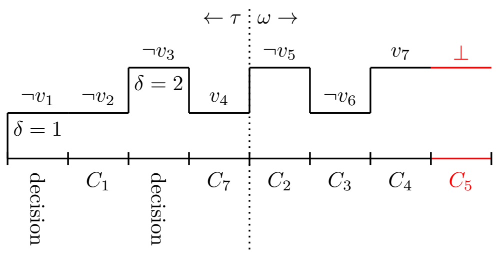


Figure 4 The clause $\neg v_3@2 \vee \neg v_4@1 \vee v_2@1$ is a missed lower implication in this example. v_2 and $\neg v_4$ are falsified at level 1, whereas $\neg v_3$ is only satisfied at level 2.

5 Soundness of Lazy Reimplication

This section proves the soundness and completeness[2] of our LSCB approach given in Algorithm 1. We note that Algorithm 1 implements strong chronological backtracking and does not miss any implication; as such, Invariant 4 holds.

▶ **Theorem 13** (Soundness of conflict analysis). *Let $C \in F$ be a conflicting clause with the partial assignment π. Then, conflict analysis in* ANALYZE(C) *from Algorithm 1 returns a conflicting clause that is implied by the clause set.*

Proof. The starting clause $D \leftarrow C$ is conflicting. At each step, D is resolved with a clause C' such that $C' = \rho(\ell)$ or $C' = \lambda(\ell)$, with $\ell \in C'$ and $\neg\ell \in D$. From the definition of ρ and λ, we have $\big(C' \setminus \{\ell\} \wedge \pi\big) \models \perp$. Therefore, the clause $D \leftarrow D \otimes_{\neg\ell} C'$ is conflicting, and implied by F, since $C' \in F$. ◀

▶ **Theorem 14** (No missed unit implication). *Algorithm 1 satisfies Invariant 9. As such, our LSCB method in Algorithm 1 does not miss unit implications.*

Proof. We prove that Invariant 9 holds for each building block of Algorithm 1.

[2] with details also in the code base of NAPSAT

BCP. Invariant 9 trivially holds at the starting state, where $\pi = \emptyset$. Further, during the propagation of one literal, Algorithm 2 ensures that for each clause $C \in F$ watched by c_1 and c_2, the following Hoare triple holds

$$\{P\}\textsc{PropagateLiterall}(\ell)\{Q\},$$

where

$$\begin{aligned} P &\equiv \neg c_1 \in \tau \Rightarrow [c_2 \in \pi \wedge [\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)]] \\ Q &\equiv \neg c_1 \in (\tau \cdot \ell) \Rightarrow [c_2 \in \pi \wedge [\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)]] \end{aligned}$$

By structural induction over the statements of Algorithm 2, we conclude that Invariant 9 is maintained by BCP.

Backtracking. During backtracking in Algorithm 3, each literal c_i is inspected: c_i is either removed from the trail π or c_i is kept. Violating Invariant 9 means that a literal c_2 from the trail is removed such that $\neg c_1 \in \tau \wedge c_2 \notin \pi$ for some clause $C = \{c_1, c_2, \ldots, c_m\}$ (since the levels are not altered). However, this case is rectified, since either $\delta(c_1) \leq \delta(c_2)$ (and then $\neg c_1$ would be removed from τ), or $\delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)$ (and then c_2 would be reimplied at level $\delta(\lambda(c_2) \setminus \{c_2\})$ and $c_2 \in \pi \wedge \delta(c_2) \leq \delta(c_1)$ would be true), or $\delta(\lambda(c_2) \setminus \{c_2\}) > \delta(c_1)$ (and then $\neg c_1$ is also backtracked). As such, backtracking in Algorithm 3 preserves Invariant 9.

Analysis. Within conflict analysis in Algorithm 4, the state of CDCL is not modified, only read. Therefore, any invariant that held before Algorithm 4 also holds after Algorithm 4.

CDCL. We finally ensure that Invariant 9 is maintained by Algorithm 1 also during its decision step and while adding a clause to the formula F. First, deciding still preserves Invariant 9, since it merely adds a non-assigned literal to the propagation queue ω. Second, after backtracking in Algorithm 1, we know by construction that the learned clause will have a single literal ℓ that is unassigned. This literal ℓ is then implied at level $\delta(D \setminus \{\ell\})$, satisfying Invariant 9 since the second watched literal c_2 is falsified at level $\delta(D \setminus \{\ell\})$. ◀

▶ **Corollary 15** (No missed conflict/implication)**.** *Our LSCB method from Algorithm 1 preserves the strong watched literal property of Invariant 4.*

Based on the results above, we conclude the soundness and completeness of LSCB.

▶ **Theorem 16** (LSCB soundness and completeness)**.** *Lazy reimplication with strong chronological backtracking from Algorithm 1 is sound and complete.*

Proof. Theorem 13 implies that clauses added to the clause set are implied by F. By induction, if ϕ is the original CNF, then if $\phi \models F$ and $F \models C$, then $\phi \models F \cup \{C\}$. Furthermore, from Corollary 15 we conclude that Invariant 4 holds.

Algorithm 1 returns unsat iff there exists a conflict at level 0; that is, there exists a set of clauses $F' \subseteq F$ such that $F' \models \bot$. As $\phi \models F$, then $\phi \models \bot$, and thus ϕ is unsatisfiable. Otherwise, Algorithm 1 returns SAT if a model τ exists such that every variable has been assigned and propagated ($\pi = \tau$). Based on Invariant 4, no conflict is possible and $\phi \models \tau$. ◀

▶ **Theorem 17** (Topological order in LSCB)**.** *The literals reimplied by the backtracking procedure of Algorithm 3 respect the topological order of the implication graph.*

Proof. The reimplied literals cannot depend on each other. Indeed, if they are reimplied, their implication level before backtracking was higher than d. Therefore, if a literal ℓ depends on a literal ℓ' in the implication graph, then $\delta(\ell) \geq \delta(\ell')$. If the missed lower implication $\lambda(\ell)$ has a level lower than d, then all literals in $\lambda(\ell) \setminus \{\ell\}$ are lower than d, and therefore were not backtracked. Therefore, since all literals are independent, they can be reimplied in an arbitrary order at the end of the trail, and still respect the topological order. ◀

6 Empirical Analysis

In this section, we discuss the implementation of Algorithm 1 in our new SAT solver NapSAT. We also integrated it in CaDiCaL and Glucose, and present our empirical results using NapSAT, CaDiCaL, and Glucose.

6.1 NapSAT for Lazy Reimplication in CDCL

We implemented our LSCB method from Algorithm 1 in the new SAT solver NapSAT. Our NapSAT tool is a CDCL solver using the watcher list scheme [19] with blocker literals [9]. NapSAT supports the backtracking variants of NCB, WCB, RSCB, and LSCB at runtime. In chronological backtracking, the backtracking scheme is purely chronological, that is, NapSAT always backtracks to one level before the conflict (unlike CaDiCaL). NapSAT uses the VSIDS decision heuristic [19] with the agility restart strategy [3] and root-level clause elimination [7]. NapSAT is available at `https://github.com/RobCoutel/NapSAT` and consists in a total of ~5.800 loc, among which the core of the solver represents ~1.500 loc.

Blocker literals in NapSAT. Blocker literals are useful to reduce the number of pointer dereferencing of the literal pointer [9]. If the blocker b is assigned at a level higher than the literal ℓ being falsified, then it might get backtracked before ℓ and a conflict might be missed. Invariant 9 can therefore be weakened, while still ensuring that no unit implication is missed.

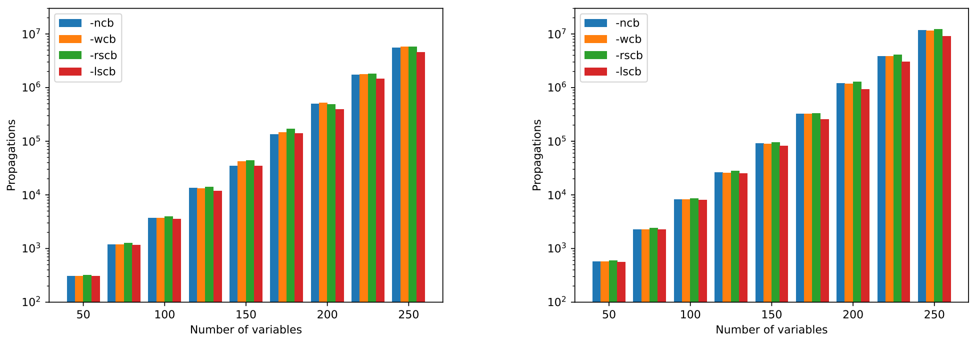
▶ **Invariant 18** (Lazy backtrack compatible watched literals with blocker literals). *For each clause $C \in F$ watched by the two distinct literals c_1, c_2 and with blocker b, we have*

$$\neg c_1 \in \tau \Rightarrow \Big(c_2 \in \pi \wedge \big(\delta(c_2) \leq \delta(c_1) \vee \delta(\lambda(c_2) \setminus \{c_2\}) \leq \delta(c_1)\big)\Big) \\ \vee \Big(b \in \pi \wedge \delta(b) \leq \delta(c_1)\Big)$$

The eager update of blocking literals is in essence similar to strategies that aggressively update watched literals during BCP [17].

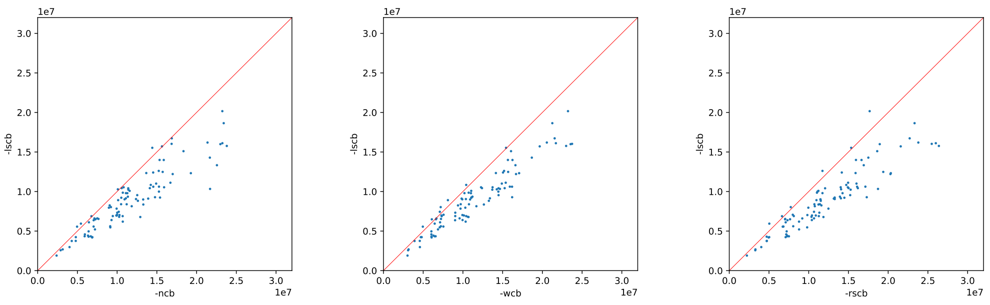
Experiments. Figure 5 shows the average total number of propagations of NapSAT on the 3-SAT uniform random problems from SATLIB [16]. Our LSCB method from Algorithm 1, indicated via `-lscb`, performs better than the other backtracking versions of NapSAT, both for satisfiable and unsatisfiable instances. Figure 6 shows more details. In particular, it shows the total number of propagations of each unsatisfiable problem with 250 variables. It shows that LSCB consistently has fewer propagations than NCB, WCB, and RSCB.

We acknowledge that the number of propagations alone is not always representative of the real performance of a SAT solver, since propagation in LSCB is slightly more expensive than in NCB or WCB. However, the number of propagations in NapSAT indicates the impact of



(a) Satisfiable instances. **(b)** Unsatisfiable instances.

Figure 5 Average total number of propagations performed by NapSAT on the SATLIB 3-SAT random problem, clustered by the number of variables, and backtracking technique employed.



(a) LSCB vs. NCB. **(b)** LSCB vs. WCB. **(c)** LSCB vs. RSCB.

Figure 6 Total number of propagations of NapSAT for each unsatisfiable Uniform Random 3-SAT problem from SATLIB. The red line is the equality line. Marks under the equality line favour our new approach.

missed lower implications. For example, Figure 5 shows that restoring the trail with RSCB might not be worth finding the missed lower implications in the random 3-SAT benchmarks; yet, reimplying literals lazily significantly reduces the total number of propagations.

6.2 Integration of LSCB in CaDiCaL and Glucose

LSCB in CaDiCaL. We implemented our LSCB approach from Algorithm 1 in CaDiCaL [4], the baseline solver of the hack track of the SAT Competition. Thanks to the built-in model-based tester Mobical, the most effort came with ensuring that we have implemented correctly Invariant 4: CaDiCaL does not require watching the literals of two highest levels when the clause is propagating. This, however, requires iterating over the clause to find the propagation level, which we do not need.

We remark that we did not change the default backjumping policy of CaDiCaL: when backjumping from more than 100 levels occurs (following the value implemented in CaDiCaL), we resort to backtracking (going one level back). Otherwise, an algorithm similar to trail reuse for restarts is used to decide how many levels should be kept. Unlike the version implemented in NapSAT, in CaDiCaL we store the missed level instead of checking the level of the MLI each time we need the level.

Table 2 Number of solved instances by different variants of strong backtracking on the SC2023 competition, using a 5.000 s timeout.

CaDiCaL version	solved	PAR-2 ($\times 10^3$)
base-CaDiCaL = RSCB	**248**	**4.09**
LSCB, Analyze-2 and minimization	246	4.16
ESCB	245	4.16
LSCB and Analyze-2	246	4.19
NCB	247	4.19
LSCB and Analyze-1	242	4.24

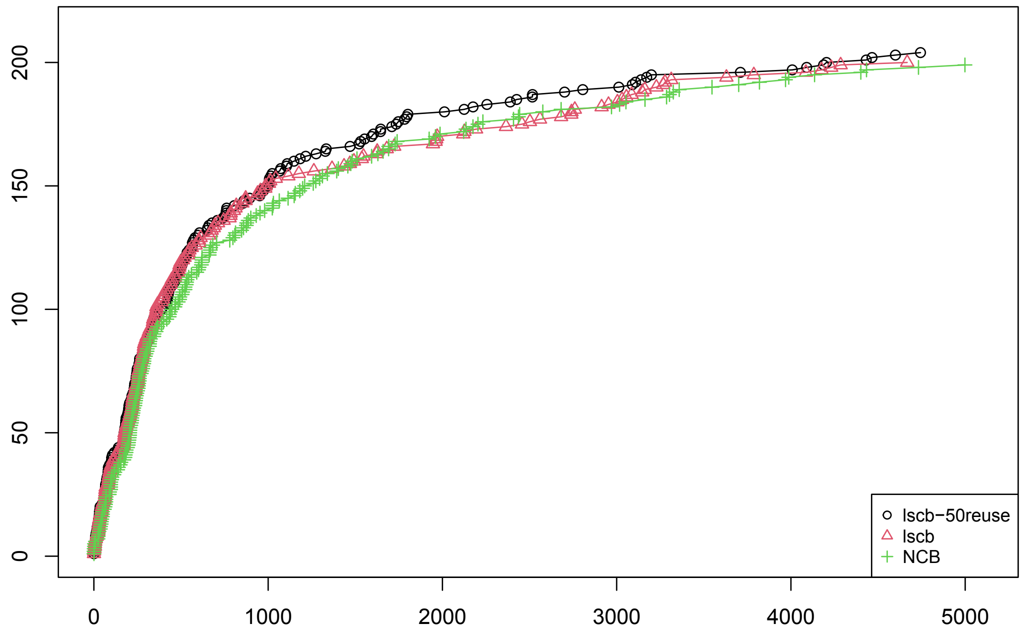
We tested various configurations, as summarized in Table 2, on the bwForCluster Helix with AMD Milan EPYC 7513 CPUs, using a memory limit of 16 GB RAM on the problems from the SAT Competition. Overall, we can see there is little difference between the considered configurations. In particular, the performance difference between WCB and NCB is limited, making it unclear if chronological backtracking is important. However, similar to the original CaDiCaL implementation [18], on the benchmarks from the SAT Competition 2018, there is an improvement from WCB over NCB. Our intuition is that chronological backtracking is especially useful when the decision heuristic is picking the wrong literals finding conflicts late instead of early (like finding a new unit at level 500 instead of level 1). The decision heuristics seem to perform worse on the 2018 benchmarks, while this did not seem to have happened since.

While the results in NapSAT seem to indicate a large decrease in the number of propagations, three factors mitigate this effect in CaDiCaL: (i) propagating a literal ℓ a second time as in RSCB is cheaper than propagating it for the first time. Most clauses remaining in the watch list of ℓ will already be satisfied and are faster to check. (ii) RSCB allows to use of blocking literals more loosely. There is no need to compare the level of the blocking literal and the propagated one, making them more potent. (iii) Searching for a replacement literal is slightly more expensive in LSCB since we need to record the highest literal in the clause.

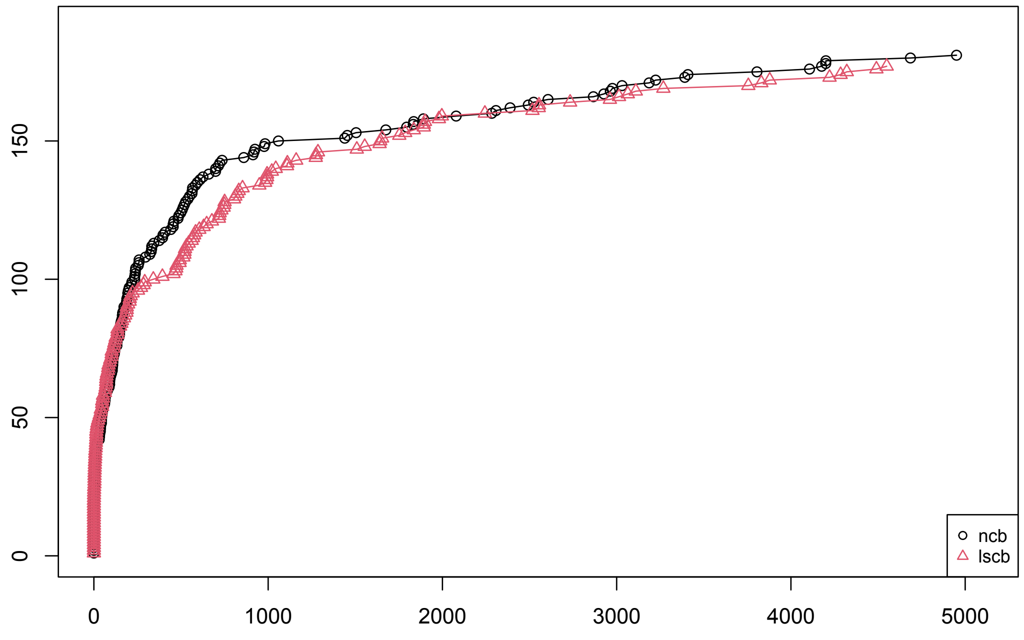
In a context where propagations are more expensive, such as SMT or user-propagators [8], these considerations might weigh less on the overall performance of the solver. We will investigate these applications for future work.

LSCB in Glucose. We also implemented our Algorithm 1 for LSCB into the latest version of Glucose [1], the SAT solver that pioneered the LBD heuristic for the usefulness of clauses (only without the minimization part). This is the only solver where we implemented the LSCB without any existing CB in the code. The entire diff (including new logging information and more assertions) is less than 1.000 lines. Our actual implementation of LSCB in Glucose is very close to our abstract Algorithm 1, because the blocker literal is always exactly the other watched literal. We use the simple heuristic to backtrack one level if jumping back more than 100, otherwise use the normal backjumping. We did not change the heuristic to block restarts [2], which is based on the trail length.

While running Glucose with LSCB on the SAT Competition 2023 (Fig. 7b), we observed worse performance. Interestingly, this is mostly due to one family of benchmarks, `SC23_Timetable`, that perform much worse with strong backtracking (but are solved eventually). On the 2018 benchmarks again (Fig. 7a), we observed a slight performance improvement when using Glucose with LSCB and it seems to be better to trigger chronological backtracking more often.



(a) Glucose variants in the SAT Competition 2018.



(b) Glucose variants in the SAT Competition 2023.

Figure 7 CDF of the different Glucose (without strategy adapting) versions. The constant indicates when chronological backtracking is triggered instead of backjumping: We apply chronological backtracking when NCB would require jumping back more than 100 levels by default, like in CaDiCaL. In the SAT Competition 2018, the version that triggers chronological backtracking for more than 50 levels performs best.

7 Conclusion

We introduce a lazy reimplication procedure to be used in CDCL with (variants of) chronological backtracking. We particularly focus on the definitions of weak chronological backtracking (WCB), restoring strong chronological backtracking (RSCB), eager strong chronological backtracking (ESCB), and lazy strong chronological backtracking (LSCB). Our invariant properties on these backtracking variants exploit watched literals. We prove that our approach of lazy reimplication in strong chronological backtracking (LSCB) yields a sound and complete SAT solving method. Our implementation in NapSAT, and its integration with CaDiCaL and Glucose, gives practical evidence that LSCB is significantly easier to implement than ESCB, while also propagating fewer literals than RSCB, and providing better guarantees than WCB.

In the future, we intend to extend our LSCB method to reason over virtual literal levels, that is, levels of missed lower implications if such a clause is detected. We believe such an extension would allow to converge closer to the guarantees of ESCB, while mitigating both algorithmic complexities and reimplication costs. Further, we will explore the integration of chronological backtracking variants in the context of SMT, as a robust approach to handling arbitrary incremental clauses and expensive theory propagations.

References

1 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, pages 399–404, 2009. URL: `http://ijcai.org/Proceedings/09/Papers/074.pdf`.

2 Gilles Audemard and Laurent Simon. On the Glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. `doi:10.1142/S0218213018400018`.

3 Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In *SAT*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008. `doi:10.1007/978-3-540-79719-7_4`.

4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel und Vijay Ganesh, editor, *Computer Aided Verification - 36th International Conference, CAV 2024, Paris, France, July 24-27, 2024*, LNCS. Springer, 2024. To appear.

5 Armin Biere, Mathias Fleury, and Florian Pollitt. CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023. In *SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 14–15. University of Helsinki, 2023.

6 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. `doi:10.3233/FAIA336`.

7 Armin Biere, Matti Järvisalo, and Bejamin Kiesl. Preprocessing in 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*, pages 391–435. IOS Press, 2nd edition edition, 2021.

8 Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. User-propagators for custom theories in SMT solving. In David Déharbe and Antti E. J. Hyvärinen, editors, *Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022*, volume 3185 of *CEUR Workshop Proceedings*, pages 71–79. CEUR-WS.org, 2022. URL: `https://ceur-ws.org/Vol-3185/extended6630.pdf`.

9 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache Conscious Data Structures for Boolean Satisfiability Solvers. *JSAT*, 6(1-3):99–120, 2009. URL: https://satassociation.org/jsat/index.php/jsat/article/view/71.

10 Robin Coutelier. RobCoutel/NapSAT. Software, swhId: swh:1:dir:1308f5717399bd09dcad2de805cc42eaa5504854 (visited on 2024-08-05). URL: https://github.com/RobCoutel/NapSAT.

11 Robin Coutelier et al. Chronological vs. Non-Chronological Backtracking in Satisfiability Modulo Theories. Master's thesis, Université de Liège, Liège, Belgique, 2023.

12 Mathias Fleury. arminbiere/cadical. Software, swhId: swh:1:dir:eaf1bada31f3142996582c25a7df2118e7cacc98 (visited on 2024-08-05). URL: https://github.com/arminbiere/cadical/tree/strong-backtrack.

13 Mathias Fleury. m-fleury/glucose. Software, swhId: swh:1:dir:fc5f0bd80c6a9e9412c5a3f3fcde96bf17a36147 (visited on 2024-08-05). URL: https://github.com/m-fleury/glucose.

14 Robin Coutelier Pascal Fontaine. ModularIT solver. Accessed March 2024. URL: https://gitlab.uliege.be/smt-modules/.

15 Randy Hickey and Fahiem Bacchus. Trail saving on backtrack. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2020. doi:10.1007/978-3-030-51825-7_4.

16 Holger H Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *Sat*, 2000:283–292, 2000.

17 Norbert Manthey. Watch Sat and LTO for CaDiCaL. In *SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 10–11. University of Helsinki, 2023.

18 Sibylle Möhle and Armin Biere. Backing Backtracking. In *SAT*, volume 11628 of *LNCS*, pages 250–266. Springer, 2019. doi:10.1007/978-3-030-24258-9_18.

19 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

20 Alexander Nadel. Introducing Intel(R) SAT Solver. In *SAT*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.SAT.2022.8.

21 Alexander Nadel and Vadim Ryvchin. Chronological Backtracking. In *SAT*, volume 10929 of *LNCS*, pages 111–121. Springer, 2018. doi:10.1007/978-3-319-94144-8_7.

22 Florian Pollitt. Cadical 1.9.4. Accessed March 2024. URL: https://github.com/arminbiere/cadical/tree/reimply-branch.

23 João P. Marques Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999. doi:10.1109/12.769433.

24 Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In *SAT*, volume 5584 of *LNCS*, pages 237–243. Springer, 2009. doi:10.1007/978-3-642-02777-2_23.

New Lower Bounds for Polynomial Calculus over Non-Boolean Bases

Yogesh Dahiya
The Institute of Mathematical Sciences (A CI of Homi Bhabha National Institute), Chennai, India

Meena Mahajan
The Institute of Mathematical Sciences (A CI of Homi Bhabha National Institute), Chennai, India

Sasank Mouli
Indian Institute of Technology Indore, India

Abstract

In this paper, we obtain new size lower bounds for proofs in the Polynomial Calculus (PC) proof system, in two different settings.

- When the Boolean variables are encoded using ± 1 (as opposed to $0, 1$): We establish a lifting theorem using an asymmetric gadget G, showing that for an unsatisfiable formula F, the lifted formula $F \circ G$ requires PC size $2^{\Omega(d)}$, where d is the degree required to refute F. Our lower bound does not depend on the number of variables n, and holds over every field. The only previously known size lower bounds in this setting were established quite recently in [Sokolov, STOC 2020] using lifting with another (symmetric) gadget. The size lower bound there is $2^{\Omega((d-d_0)^2/n)}$ (where d_0 is the degree of the initial equations arising from the formula), and is shown to hold only over the reals.
- When the PC refutation proceeds over a finite field $\mathbb{F}_p$ and is allowed to use extension variables: We show that there is an unsatisfiable $\mathsf{AC}^0[p]$ formula with N variables for which any PC refutation using $N^{1+\epsilon(1-\delta)}$ extension variables, each of arity at most $N^{1-\epsilon}$ and size at most N^c, must have size $\exp(\Omega(N^{\epsilon\delta}/\mathsf{poly}\log N))$. Our proof achieves these bounds by an XOR-ification of the generalised $\mathrm{PHP}_n^{m,r}$ formulas from [Razborov, CC 1998].
  The only previously known lower bounds for PC in this setting are those obtained in [Impagliazzo-Mouli-Pitassi, CCC 2023]; in those bounds the number of extension variables is required to be sub-quadratic, and their arity is restricted to logarithmic in the number of original variables. Our result generalises these, and demonstrates a tradeoff between the number and the arity of extension variables. Since our tautology is represented by a small $\mathsf{AC}^0[p]$ formula, our results imply lower bounds for a reasonably strong fragment of $\mathsf{AC}^0[p]$-Frege.

2012 ACM Subject Classification Theory of computation → Proof complexity

Keywords and phrases Proof Complexity, Polynomial Calculus, degree, Fourier basis, extension variables

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.10

Funding *Meena Mahajan*: Supported in part by the J. C. Bose Fellowship of SERB, ANRF.

1 Introduction

Propositional proof complexity is the field of study of the complexity of proofs for tautological Boolean formulae. Cook and Reckhow [6] introduced this area in their seminal work with the ultimate goal of resolving the question of NP versus coNP using upper/lower bounds for stronger and stronger proof systems. Polynomial Calculus (PC) is one such propositional proof system that has received wide attention since its introduction by Clegg, Edmonds and Impagliazzo [5]. Degree lower bounds for PC and its variant PCR (PC with Resolution) have been proved starting with the work of Razborov [17], followed by a long series of works [12, 4, 1, 14, 7]. These translated to size lower bounds through a size-degree connection established in [5, 12]. Despite these works showing that we have a reasonable understanding of PC, there has been little progress towards lower bounds for the stronger system $\mathsf{AC}^0[p]$-Frege, which was one of the main motives for defining PC. Therefore, this indicates that we have to look at systems stronger than PC in order to get new insights.

Grigoriev and Hirsch [8] introduced one such system called constant-depth PC, where in addition to the rules of PC we allow extension variables of constant depth to be introduced and used as new variables. They showed that this system simulates $\mathsf{AC}^0[p]$-Frege (at a proportional depth), making it a suitable target for lower bounds. However, Raz and Tzameret [16] showed that this system is already powerful enough to simulate the proof system Cutting Planes (which deals with integer linear inequalities) with polynomially bounded coefficients. Finally, Impagliazzo, Mouli and Pitassi [10] showed that this system can simulate at a fixed constant depth $\mathsf{AC}^0[q]$-Frege for any prime modulus q, Cutting Planes and the semialgebraic proof system sum-of-squares SOS with unbounded coefficients, and can also simulate TC^0-Frege at a proportional depth. This implies that general lower bounds for this system are much harder than lower bounds for $\mathsf{AC}^0[p]$-Frege.

The simplest subsystem of the above for which size lower bounds were unknown until recently is PC over ± 1 variables instead of $\{0, 1\}$. The switch from the latter basis to the former can be made using an affine transformation, which preserves degree lower bounds. However, known techniques based on the size-degree connection fail over this basis since they rely on terms vanishing when variables are set to zero. Moreover, the Tseitin tautologies require large PC degree but have small PC size over ± 1, precluding the existence of such a generic connection. Sokolov [18] recently showed how to go past this barrier and proved size lower bounds for PC over ± 1 variables. Impagliazzo, Mouli and Pitassi [11] improved and generalized these bounds to PC over finite fields $\mathbb{F}_p$ with a sub-quadratic number of extension variables, where each extension variable depends on $O(\log n)$ original variables.

Our Results

In this work, we extend the results of both [18] and [11]. For our first result, we show a generic degree-to-size lifting result for PC over ± 1 basis ($\mathsf{PC}_{\{\pm 1\}}$).

▶ **Theorem 1.1.** *Let F be an unsatisfiable formula over variables $x_1 \cdots x_n$, with a polynomial encoding of degree d_0, which requires degree $d > d_0$ to refute in PC. Let Ind denote the one-bit indexing gadget. Let $F \circ \mathit{Ind}$ be the formula obtained by replacing each x_i by $\mathit{Ind}(w_{i0}, w_{i1}, w_{i2})$ for a fresh set of variables $w_{i0}, w_{i1}, w_{i2} \in \{\pm 1\}$. Then $F \circ \mathit{Ind}$ requires size $2^{\Omega(d)}$ to refute in PC over ± 1 basis.*

Sokolov showed such a lifting result for the SOS proof system, using a symmetric gadget with certain properties (e.g. majority). By showing that SOS can simulate $\mathsf{PC}_{\{\pm 1\}}$ over the reals (with respect to both size and degree), similar to the findings in [3] for the $\{0, 1\}$ basis, Sokolov also obtained a SOS degree to $\mathsf{PC}_{\{\pm 1\}}$ size lift over $\mathbb{R}$. In his concluding remarks, Sokolov posed the question of directly proving a degree-to-size lifting result for $\mathsf{PC}_{\{\pm 1\}}$, irrespective of the field. Our result addresses this question, offering a lifting result for $\mathsf{PC}_{\{\pm 1\}}$ over any field $\mathbb{F}$ using a one-bit indexing gadget.

Furthermore, Sokolov's $\mathsf{PC}_{\{\pm 1\}}$ size lower bounds over $\mathbb{R}$ for lifted formulas are of the form $2^{\Omega((d-d_0)^2/n)}$ (where n is the number of variables, d_0 is the degree of the initial equations arising from the unlifted formula, and d is the SOS degree lower bound for unlifted formula), yielding meaningful results only when $d = \omega(\sqrt{n})$. In contrast, our result offers meaningful $\mathsf{PC}_{\{\pm 1\}}$ size lower bounds for lifted formulas as long as the unlifted formulas have a superconstant PC degree lower bound. For instance, the graph version of the Pigeonhole

Principle (GPHP) based on a sufficiently expanding bipartite graph has a constant degree proof over SOS [9] but has a degree of $\Omega(n)$ over PC [1, 14]. Hence, using Sokolov's lifting theorem with GPHP as an unlifted formula yields nothing, whereas our result shows that GPHP lifted with a one-bit indexing gadget has exponential size in $\mathsf{PC}_{\{\pm 1\}}$ irrespective of the field. This also gives a straightforward exponential size lower bound for $\mathsf{PC}_{\{\pm 1\}}$, a result only recently proven in Sokolov's work. Lastly, we believe that our result is arguably simpler to prove. This result is inspired by the work of Krause and Pudlák [13].

As corollaries of this result, we also obtain (1) a size lower bound in $\mathsf{PC}_{\pm 1}$ for random 3-CNF formulas lifted with one bit indexing, Corollary 3.8, (2) an arguably simpler proof of the size separation between $\mathsf{PC}_{\{\pm 1\}}$ and $\mathsf{SOS}_{\{\pm 1\}}$, Corollary 3.9, and (3) an improved version of the degree-to-size lifting for $\mathsf{SOS}_{\{\pm 1\}}$, Theorem 3.10.

In our next result, we strengthen the lower bounds of [11] for PC over finite fields to handle a sub-quadratic number of extension variables, each of which is of polynomial size and depends on a polynomial fraction of the original variables. While [11] showed lower bounds under the same setting for a sub-quadratic number of extension variables, they restricted the arity of these extension variables, i.e., the number of original variables they can depend on, to be logarithmic in the number of original variables. Thus, we have an exponential improvement over the result of [11] in the *arity* of the extension variables provided the extension axioms remain small. Moreover, the constraint of polynomial-sized extension axioms in our result is also implicitly present in [11], as they restrict the arity of extension axioms to be logarithmic in the number of original variables, thereby only allowing polynomial-sized extension axioms.

▶ **Theorem 1.2.** *For every $N > 0$ large enough, any $1 > \epsilon, \delta > 0$, constant $c > 0$, and prime p, there exists a tautology F over N variables such that any PC refutation of F over $\mathbb{F}_p$ with $N^{1+\epsilon(1-\delta)}$ extension variables, each depending on $N^{1-\epsilon}$ variables of F and of size at most N^c, requires size $\exp(\Omega(N^{\epsilon\delta}/\mathsf{poly}\log N))$.*

Related work

As mentioned earlier, our lower bounds strengthen those in [18], [11]. For Polynomial Calculus with extension variables, over the reals, stronger lower bounds have been shown in [2], but these are incomparable to our result in Theorem 1.2 as we primarily focus on finite fields.

Our Techniques

In both our results, the notion of quadratic degree introduced in [18] plays a crucial role. This is the maximum degree that can be obtained from a PC refutation by multiplying any two terms that appear in the same line; see Definition 3.2. Sokolov's insight was that it is easier to reason about this measure for a refutation, and at the same time, it does carry information about the usual degree. In particular, a refutation with low quadratic degree can be transformed into one with low degree; Lemma 3.7. An adaptation of this measure and this method was subsequently used in [11] to similarly reason about refutations using extension variables. A non-trivial part in both these is establishing that if a variable appears in the refutation but not in any axiom (other than the Boolean axiom or an extension axiom of a specific type), then it can be removed from the proof; Lemmas 3.4 and 4.13. This seems self-evident but needs to be done with care.

To prove our first result Theorem 1.1, we follow Sokolov's approach, but we employ a lift by the (asymmetric) one-bit Indexing gadget. The gadget, on variables w_0, w_1, w_2, selects the value of w_1 if $w_0 = -1$ (in this case, w_2 is irrelevant) and that of w_2 otherwise (now w_1 is irrelevant). If there is a small enough proof, then the probabilistic method guarantees the existence of an assignment to all the selector variables so that under this restriction, every high-degree quadratic term contains an irrelevant variable. The restricted refutation is in fact a refutation of the unlifted formula, but may use irrelevant variables along the way. Such irrelevant variables can be removed from the restricted proof as discussed above, and then Sokolov's transformation from low-quadratic-degree to low-degree can be employed, yielding a contradiction.

To prove our second result, we closely follow the approach of [11], which we outline in Section 4. However, to handle extension variables of large arity and polynomial size, we use a similar idea as above; composing a hard tautology with a simple gadget. The gadget we use here is just the XOR_2 gadget (the parity of two variables). We begin by considering family of restrictions where, for each XOR_2 gadget, one of the variables is assigned a random bit while the other variable remains free. Such restrictions recover the hard formula (possibly with some variables negated). Using the probabilistic method once again, it is easy to guarantee the existence of one such restriction under which every extension variable axiom, despite having a large arity, reduces to logarithmic degree due to its polynomial size. Thus the problem reduces to proving a size lower bound for the original unlifted tautology, where extension axioms are bounded by logarithmic degree. Such extension variables can be handled using the approach of [11] by further carefully chosen restrictions of small size and even smaller Hamming weight; this size lower bound is shown in Theorem 5.5. The hard formula we choose is the generalised Pigeon-Hole-Principle formula $\mathrm{PHP}_n^{m,r}$, introduced by Razborov in [17] (see Proposition 5.2), and we show that applying such small restrictions preserves the degree hardness shown by Razborov. The system of polynomials over $\mathbb{F}_p$ underlying our tautology is not a translation of a small CNF formula (as in [11]). Nevertheless, they can be represented by small size, low depth $\mathsf{AC}^0[p]$ circuits. Therefore, our lower bounds still imply lower bounds for (the corresponding fragment of) $\mathsf{AC}^0[p]$-Frege.

We note here that the notions of Quadratic degree and the associated operation *Split* which reduces it were introduced by Sokolov [18] and generalized in Impagliazzo, Mouli, Pitassi [11]. Since we extend results of both works, we use two different but very related notions of Quadratic degree and *Split* in this paper: in Section 3 we use the notions from [18] and in Sections 4 and 5 we use the notions from [11].

Organisation of the Paper

In Section 2, we include the basic relevant definitions of the proof systems. In Section 3 we describe the setting where the encoding is over ± 1, and prove Theorem 1.1. In Section 4, we consider the setting where extension variables are used, and describe all the facts and results from [11] that we crucially use. In Section 5 we prove Theorem 5.5 and Theorem 1.2.

2 Preliminaries

We follow the notation from [18, 11].

▶ **Definition 2.1** (Polynomial Calculus/Polynomial Calculus Resolution). *Let $\Gamma = \{P_1 \dots P_m\}$ be an unsolvable system of polynomials in variables $\{x_1 \dots x_n\}$ over $\mathbb{F}$. A* PC *(Polynomial Calculus) refutation of Γ is a sequence of polynomials $\{R_1 \dots R_s\}$ where $R_s = 1$, and for every $\ell \in [s]$, R_ℓ is either a polynomial from Γ, or is obtained from two previous polynomials R_j, R_k, $j, k < \ell$, by one of the following derivation rules:*

$R_\ell = \alpha R_j + \beta R_k$ for α, $\beta \in \mathbb{F}$

$R_\ell = x_i R_k$ for some $i \in [n]$

The size of the refutation is $\sum_{\ell=1}^{s} |R_\ell|$, where $|R_\ell|$ is the number of monomials in the polynomial R_ℓ. The degree of the refutation is $\max_\ell deg(R_\ell)$.

A PCR *(Polynomial Calculus Resolution) refutation is a* PC *refutation over the set of Boolean variables $\{x_1 \dots x_n, \bar{x}_1 \dots \bar{x}_n\}$ where $\{\bar{x}_1 \dots \bar{x}_n\}$ are twin variables of $\{x_1 \dots x_n\}$. That is, over the $\{0,1\}$ encoding, the equations $x_i^2 - x_i = 0$, $\bar{x_i}^2 - \bar{x}_i = 0$ and $x_i + \bar{x}_i - 1 = 0$ are treated as axioms. Similarly, over the ± 1 encoding, the equations $x_i^2 - 1 = 0$, $\bar{x_i}^2 - 1 = 0$ and $x_i\bar{x}_i + 1 = 0$ are treated as axioms.*

In the literature, the terms PC and PCR are often used interchangeably. The notion of degree is the same in both, but size in PC with the $\{0,1\}$ encoding of Boolean variables can be much larger than in PCR. Throughout this paper, we say PC but really mean PCR. In particular, our size lower bounds are for PCR.

Note that the minimal degree required to refute a formula is independent of whether Boolean variables are encoded over $\{0,1\}$ or over ± 1. However, the minimal size crucially depends on the encoding. As is well known [4], suitable Tseitin formulas require degree n, hence they require size $\exp(\Omega(n))$ over the $\{0,1\}$ basis using size-degree connection, but have linear-size refutations over ± 1.

As is standard, we work in the ideal modulo the Boolean axioms, and hence the polynomials in all lines are multilinear in the original variables. Technically, on deriving a higher degree term, it has to be cancelled by using suitable multiples of the Boolean axiom; however, these steps do not significantly alter the size or degree of the refutation.

▶ **Definition 2.2** (PC plus Extension Axioms). *Let $\Gamma = \{P_1 \dots P_m\}$ be a set of polynomials in variables $\{x_1 \dots x_n\}$ over a field $\mathbb{F}$, with no common zero. The polynomials in Γ are referred to as the (initial) axioms. Let $\mathbf{z} = z_1 \dots z_M$ be new extension variables with corresponding extension axioms $z_j - Q_j(x_1 \dots x_n)$. A* PC + Ext *(*PC *plus extension) refutation of Γ with M extension axioms $Ext = \{z_j - Q_j(x_1 \dots x_n) \mid j \in [M]\}$ is a* PC *refutation of the set of polynomials $\Gamma' = \{P_1 \dots P_m, z_1 - Q_1 \dots z_M - Q_M\}$. The size of the refutation is the total size of all lines in the refutation, including the polynomials in Γ' (where the size of a line $P \in \Pi$ is the number of monomials in P). The degree of the refutation is the maximum degree of any line in the refutation or in Γ'.*

Similar to [11], our notion of extension variables is not recursive in the sense that new extension variables cannot be defined as functions of existing ones. Our extension variables are only allowed to depend on the original variables of the tautology.

We also consider the Sum-of-Squares (SOS) proof system, which is a semi-algebraic proof system. While algebraic proof systems (like PC) are defined over any field and are based on polynomial equalities, semi-algebraic proof systems are defined only over $\mathbb{R}$ and are based on polynomial inequalities. SOS serves as an analogue to the algebraic Nullstellensatz proof system (a static version of PC), albeit for polynomial inequalities over the reals, contrasting Nullstellensatz's treatment of polynomial equalities over arbitrary fields. Similar to Nullstellensatz, SOS is a static proof system.

We specifically consider SOS over Boolean variables taking values in $\{+1, -1\}$. Formally,

▶ **Definition 2.3** (Sum-of-Squares over $\{\pm 1\}$ Basis). *Let $\Gamma = \{f_1 = 0, \dots, f_m = 0; h_1 \geq 0, \dots, h_s \geq 0\}$ be an unsolvable system of polynomial equalities and inequalities over Boolean variables $\{x_1, \dots, x_n\}$ taking values in $\{+1, -1\}$. Let $\mathcal{R} = x_1^2 - 1 = 0, \dots, x_n^2 - 1 = 0$ be the range axioms enforcing x_i's to be ± 1. A* $\mathsf{SOS}_{\pm 1}$ *(Sum-of-Squares over $\{\pm 1\}$ basis) refutation of Γ is an explicit list of real polynomials $(q_0, \dots, q_s; p_1, \dots, p_m; r_1, \dots, r_n)$ such that*

$$q_0 + \sum_{i=0}^{s} q_i h_i + \sum_{i=1}^{m} p_i f_i + \sum_{i=1}^{n} r_i(x_i^2 - 1) = -1.$$

and for each $i \in \{0\} \cup [s]$, q_i is a sum of squares of polynomials.

The degree of the SOS *refutation is* $\max(deg(q_0), \max_i(deg(q_i) + deg(h_i)), \max_i(deg(p_i) + deg(f_i)))$*. The size of the* SOS *refutation is* $|q_0| + \sum_{i=0}^{s}(|q_i| + |h_i|) + \sum_{i=1}^{m}(|p_i| + |f_i|)$*, where* $|p|$ *is the number of monomials in the polynomial* p*.*

To define the degree and size, we have omitted the consideration of terms involving range axioms, as they do not significantly affect the degree or size of the proof. Henceforth, we will proceed under the assumption that all computations are performed modulo the ideal generated by the range axioms. Consequently, all polynomials involved are treated as multilinear. Furthermore, given our focus on the ± 1 basis for SOS, we have assumed the absence of twin variables, as the variable $\bar{x}$ can be readily substituted with $-x$.

3 PC size lower bounds over ± 1 by lifting with one-bit indexing

In this section we prove that if a tautology F requires PC degree d, then the tautology F' obtained by lifting each variable in F with a one-bit indexing gadget (over a fresh set of variables) requires PC size $2^{\Omega(d)}$ over ± 1.

▶ **Definition 3.1** (One-bit indexing gadget). *Let* w_0, w_1, w_2 *be variables taking values in* $\{\pm 1\}$*. The function* $Ind(w_0, w_1, w_2)$ *is defined as follows:* $Ind(-1, w_1, w_2) = w_1$ *and* $Ind(1, w_1, w_2) = w_2$*. We call* w_0 *the selector variable and* w_1, w_2 *the data variables of the gadget.*

Lifting a Boolean formula F by this gadget for each variable means introducing three fresh variables w_0, w_1, w_2 corresponding to each variable w in F, and replacing each occurrence of w in F with an expression equivalent to $(1-w_0)w_1/2+(1+w_0)w_2/2$, and adding the Boolean axioms $w_j^2 = 1$ for $j = 0, 1, 2$. (Note that if F is in CNF with narrow (logarithmic width) clauses, then the lifted formula is also expressible in CNF with only polynomial blowup.)

Our idea is to consider refutations of F lifted by the indexing gadget, then apply a restriction to the selector variables yielding a refutation of F with low quadratic degree, and hence obtain a small degree refutation of F. The quadratic degree of a refutation is defined in [18] using the notion of lazy representations of polynomials, and is rephrased below:

▶ **Definition 3.2** (Quadratic set, Quadratic degree, Quadratic terms over ± 1; taken from [18], Section 3.2). *Given a proof* Π *over* ± 1 *variables, the Quadratic set of* Π*, denoted* $\mathcal{Q}(\Pi)$*, is the set of pairs of terms* $\mathcal{Q}(\Pi) = \{(t_1, t_2) \mid t_1, t_2 \in P \text{ for some line } P \in \Pi\}$*.*
Denote by $\mathcal{QT}(\Pi)$ *the set of quadratic terms* $\{t_1 t_2 \mid (t_1, t_2) \in \mathcal{Q}(\Pi)\}$*, where the product is modulo the axioms* $x_i^2 = 1$*.*
The Quadratic degree of Π *is the max degree of a term in* $\mathcal{QT}(\Pi)$*.*
Informally, Quadratic degree is the max degree of the square of each line (before cancellations).

When we apply the chosen restriction to the selector variables in $F \circ \text{Ind}$, the irrelevant variables no longer appear in any of the axioms (except the axioms $x_i^2 = 1$; we work modulo those anyway). However, they may still appear in the refutation, and we need to eliminate them. For this, we use the Split operation introduced in [18].

▶ **Definition 3.3** (Split operation over x [18], Section 5.4). *Given a proof* $\Pi = (P_1, P_2, \ldots, P_t)$ *and a variable* $x \in \{\pm 1\}$*, each line* P_i *of* Π *is of the form* $P_{i,1}x + P_{i,0}$*, where* $P_{i,1}, P_{i,0}$ *do not contain* x*. The Split operation at* x*, denoted by* $Split_x(\Pi)$*, is the sequence* Π' *with the lines* $\{P_{1,1}, P_{1,0}, P_{2,1}, P_{2,0}, \ldots, P_{t,1}, P_{t,0}\}$*.*

The following lemma shows that Split of a refutation is a valid refutation whenever the variable we are splitting on does not appear in any axioms except $x^2 = 1$. (That is, x has no role in the tautology we are considering, but is possible introduced along the way and then eliminated. The gadget variables rendered irrelevant by our chosen restriction are like this.) This is in fact a special case of a more general statement shown in [18], and we only need this case. For ease of reading, we include here a proof of just this special case.

▶ **Lemma 3.4.** *Suppose that Π is a proof and x is a variable that does not appear in any axioms of Π except $x^2 = 1$. Then Split$_x(\Pi)$ outputs a valid proof of the axioms of Π, with no line containing x.*

Proof. Let Π be the sequence $P_1, \ldots, P_t$. We show by induction on the line number j that both $P_{j,1}$ and $P_{j,0}$ are derivable and x-free.

If P_j is an axiom, then it is free of x. So the Split version is $P_{j,1} = 0$, $P_{j,0} = P_j$, and both these polynomials are derivable.

If $P_j = \alpha P_i + \beta P_k$ for some $i, k < j$, then $P_{j,b} = \alpha P_{i,b} + \beta P_{k,b}$ for $b = 0, 1$.

If $P_j = yP_i$ for some $i < j$ and some variable $y \neq x$, then $P_{j,b} = yP_{i,b}$ for $b = 0, 1$.

If $P_j = xP_i$ for some $i < j$, then since $x^2 = 1$ we obtain $P_{j,1} = P_{i,0}$ and $P_{j,0} = P_{i,1}$.

Thus all the lines $P_{j,b}$ are derivable and do not contain x.

Since the last line of the proof is $P_t = 1$, we have $P_{t,1} = 0$ and $P_{t,0} = P_t = 1$. Thus Split$_x(\Pi)$ derives 1 and is a valid proof from the axioms of Π. ◀

▶ Remark 3.5. It may help to visualise the Split$_x$ process as follows. Consider the case where the derivation structure underlying Π is tree-like. The tree T is rooted at P_t, and is unary-binary: linear combination nodes have two children and variable-multiplication nodes have one child. The Split$_x$ process makes two nodes P_0, P_1 for each node P of T, and ends up creating a forest with two trees T_0, T_1. The desired refutation is T_0, since $P_{t,0} = 1$ whereas $P_{t,1} = 0$. It can be seen that T_0 may also be obtained directly from T as follows: for each axiom node, if the number of edges along the path to the root labeled by multiplication with x is odd, replace the axiom by 0. (The construction above would have ended at a source node which is a P_1 copy of an axiom, and since axioms are x-free, a P_1 copy of an axiom is 0.) Then, replace each edge label $\times x$ by the label $\times 1$.

The lemma below shows that Split$_x$ removes all quadratic terms containing x from the proof, without introducing any new quadratic terms.

▶ **Lemma 3.6.** *Let $\mathcal{Q}_x(\Pi)$ be the set of pairs $(t_1, t_2) \in \mathcal{Q}(\Pi)$ such that $x \in t_1t_2$, and let $\mathcal{QT}_x(\Pi)$ be the corresponding set of quadratic terms.*
If $(t_1, t_2) \in \mathcal{Q}(\text{Split}_x(\Pi))$, then t_1 and t_2 are both x-free, and at least one of (t_1, t_2), (t_1x, t_2x), is in $\mathcal{Q}(\Pi)$. Thus $\mathcal{QT}(\text{Split}_x(\Pi)) \subseteq \mathcal{QT}(\Pi) \setminus \mathcal{QT}_x(\Pi)$.

Proof. Consider a pair $(t_1, t_2) \in \mathcal{Q}(\text{Split}_x(\Pi)$. That t_1, t_2 are x-free follows from Lemma 3.4. The pair (t_1, t_2) is contributed to $\mathcal{Q}(\text{Split}_x(\Pi)$ by P_b for some line $P = xP_1 + P_0$ of Π and some $b \in \{0, 1\}$. If P_0 contributes the pair, then P also contributes the pair to $\mathcal{Q}(\Pi)$. If P_1 contributes the pair, then P contributes the pair (t_1x, t_2x) to $\mathcal{Q}(\Pi)$. ◀

Finally, we note below that a proof with low quadratic degree can be transformed into a proof of low (usual) degree. This lemma is proved in [18], Lemma 3.6 using the notion of lazy representation of polynomials. For completeness, we include here a very similar proof but without explicitly using this notion.

▶ **Lemma 3.7** ([18], Lemma 3.6)**.** *Let Π be a refutation of a set of axioms F of degree d_0 with Quadratic degree at most d. Then there exists a refutation Π' of F with (usual) degree at most $2\max(d, d_0)$.*

Proof. Let $\Pi = \{P_j\}_j$. Now, consider $\Pi' = \{P'_j\}_j$ where $P'_j = t_j P_j$, with each $t_j \in P_j$ carefully selected. Since the degree of $t_j P_j$ is bounded by the Quadratic degree of P_j, every line in Π' is of degree at most d. However, Π' is not an immediate valid refutation of F, but it can be transformed into one. We will show that each line of Π' can be derived from previous lines and axioms of F in degree at most $2\max(d, d_0)$, completing the proof. We proceed by induction on line number j.

If P_j is an axiom, then we set t_j to be an arbitrary term in P_j and derive $P'_j = t_j P_j$ in degree $2d_0$ starting from P_j.
(Note that in [18], it is claimed that this step can be derived in degree d_0. But this is not always so. For instance, if $p = x_1x_2x_3 + x_2x_3x_4 + x_3x_4x_1 + x_4x_1x_2$ has degree $d_0 = 3$, and $d = 2$, then for any term $t \in p$, tp has degree 2 but needs degree $4 > \max\{d, d_0\}$ for the derivation.)

If $P_i = xP_j$ for some $j < i$, then we select $t_i = xt_j$, and consequently, $P'_i = t_i P_i = t_j P_j = P'_j$ is derived without raising the degree.

Finally, if $P_i = P_{j_1} + P_{j_2}$, we choose t_i to be an arbitrary term in P_i and derive $P'_i = t_i P_i = t_i t_{j_1} P'_{j_1} + t_i t_{j_2} P'_{j_2}$. We argue that the degree of both $t_i t_{j_1}$ and $t_i t_{j_2}$ is at most d, and as a result, P'_i can be derived from P'_{j_1} and P'_{j_2} in degree at most $2d$, which completes the proof. To justify this assertion, let $t_i \in P_{j_1}$ without loss of generality (every term in P_i appears in either P_{j_1} or P_{j_2}). Then degree of $t_i t_{j_1}$ is bounded by the Quadratic degree of P_{j_1} and hence by d. Additionally, if $t_{j_2} \in P_i$, then the degree of $t_i t_{j_2}$ is bounded by the quadratic degree of P_i and is also bounded by d. In the case where $t_{j_2} \notin P_i$, it means that it was cancelled in the sum and therefore $t_{j_2} \in P_{j_1}$ and so degree of $t_i t_{j_2}$ is bounded by the Quadratic degree of P_{j_1} and is again bounded by d.

Thus all lines in Π' can be derived from previous lines and axioms of F in degree at most $2\max(d, d_0)$. Since the last line of Π' is 1, we get that Π' can be successfully transformed into a valid proof of F of degree $2\max(d, d_0)$. ◀

We conclude this section with a proof of Theorem 1.1, which we restate here for convenience.

▶ **Theorem 1.1.** *Let F be an unsatisfiable formula over variables $x_1 \cdots x_n$, with a polynomial encoding of degree d_0, which requires degree $d > d_0$ to refute in PC. Let Ind denote the one-bit indexing gadget. Let $F \circ$ Ind be the formula obtained by replacing each x_i by Ind(w_{i0}, w_{i1}, w_{i2}) for a fresh set of variables $w_{i0}, w_{i1}, w_{i2} \in \{\pm 1\}$. Then $F \circ$ Ind requires size $2^{\Omega(d)}$ to refute in PC over ± 1 basis.*

Proof. Let F' denote the formula $F \circ \text{Ind}$.

Towards a contradiction, let Π be a refutation of F' of size 2^{cd} for a small enough $c > 0$. An assignment ρ_i to the selector variable w_{i0} sets the gadget to one of the two data variables w_{ij}, $j \in \{1, 2\}$; we say that the other data variable is irrelevant. (If ρ_i sets $w_{i0} = -1$ then w_{i2} is irrelevant, else w_1 is irrelevant.) We construct an assignment ρ to the selector variables such that for every pair $(t_1, t_2) \in \mathcal{Q}(\Pi_{|_\rho})$ with $\deg(t_1t_2) \geq d/2$, t_1t_2 contains an irrelevant variable. The rest of the proof is simple: we apply this ρ to Π to obtain a refutation of a copy of F without irrelevant variables. However, the irrelevant variables may still appear in the proof. We then repeatedly apply Split over each irrelevant variable, to obtain a refutation Π' of F with no irrelevant variables anywhere. (By Lemma 3.4, the result of Split is a valid refutation.) Since every high-degree pair contains an irrelevant variable, and by Lemma 3.6 all pairs where the product contains an irrelevant variable are removed from the proof, Π' does not contain any high-degree pair and hence has Quadratic degree less than $d/2$. Using Lemma 3.7, we get a refutation of degree less than d of F, contradicting our assumption.

We now show the existence of ρ through a probabilistic argument. Let $t_1t_2 \in \mathcal{QT}(\Pi)$ with degree in the data variables at least $d/2$. If, for some gadget, the product t_1t_2 contains both data variables, then for any assignment ρ, t_1t_2 would contain an irrelevant variable. So without loss of generality we can assume that t_1t_2 contains only one data variable from every copy of the gadget. Now, pick a ρ uniformly at random from $\{\pm 1\}^n$; i.e. pick the data variable at random in each gadget. For a data variable in t_1t_2, the probability that it is picked is equal to $1/2$. Therefore, the probability that t_1t_2 does not contain any irrelevant variable is at most $(1/2)^{d/2}$. Since there are only 2^{cd} terms and therefore 2^{2cd} pairs in the proof, the union bound guarantees that there exists a restriction with the required property. ◀

As a corollary, we obtain an exponential size lower bound for $\mathsf{PC}_{\{\pm 1\}}$ by using any unsatisfiable CNF formula with a PC degree $\Omega(n)$ and lifting it with a one-bit indexing gadget. Specifically, by combining the lifting theorem with the result of Alekhnovich and Razborov [1] regarding the PC degree of random CNF formulas, we get the following corollary:

▶ **Corollary 3.8.** *Let ψ be a random 3-CNF formula on $m = O(n)$ clauses. Then, with high probability, any $\mathsf{PC}_{\{\pm 1\}}$-proof of $\psi \circ Ind$ has size $\exp(\Omega(n))$.*

Another corollary of our lifting result is an easy separation between SOS proof size and PC proof size over the $\{\pm 1\}$ basis, a result recently shown by Sokolov [18]. Sokolov showed that the graph version of the Pigeonhole Principle (GPHP) based on sufficiently expanding bipartite graphs has an exponential size lower bound for $\mathsf{PC}_{\{\pm 1\}}$. Together with the constant degree and polynomial size upper bound on SOS-proofs (independent of the basis) of GPHP from [9], he established an exponential separation. We can now achieve an exponential separation simply by using our lifting theorem.

▶ **Corollary 3.9.** *Let ψ be the GPHP formula on sufficiently expanding bipartite graphs, and let ψ' be its lift by the one-bit indexing gadget. Then ψ' requires exponential size over $\mathsf{PC}_{\{\pm 1\}}$, but has a polynomial size proof over $\mathsf{SOS}_{\{\pm 1\}}$.*

Proof. The GPHP formulas require $\Omega(n)$ PC degree [1, 14]. Hence, by Theorem 1.1, their lift with a one-bit indexing gadget requires exponential size over $\mathsf{PC}_{\{\pm 1\}}$.

Since GPHP has a constant degree and polynomial size proof over $\mathsf{SOS}_{\{\pm 1\}}$, lifting it with a one-bit indexing gadget will still yield a polynomial size proof. ◀

A proof similar to the PC lifting theorem also works for the SOS proof system, where given a refutation $\sum_i p_i f_i + \sum_j q_j^2 = -1$ of axioms $f_i = 0$, the operation Split at an irrelevant variable x is defined (following [18]) as the refutation obtained by averaging the values of p_i and q_i^2 at $x = 1$ and $x = -1$ (this is a valid refutation since x is irrelevant). As a result, we also obtain a degree-to-size lifting theorem for $\mathsf{SOS}_{\pm 1}$ for the one-bit indexing gadget.

▶ **Theorem 3.10.** *Let $\Gamma = \{f_1 = 0, \ldots, f_m = 0; h_1 \geq 0, \ldots, h_s \geq 0\}$ be an unsolvable system of polynomial equalities and inequalities of degree d_0 over (± 1)-valued Boolean variables $\{x_1, \ldots, x_n\}$. Let Ind denote the one-bit indexing gadget. If $d > d_0$ is the minimal degree of an $\mathsf{SOS}_{\pm 1}$ refutation of Γ, then any $\mathsf{SOS}_{\pm 1}$ refutation of $\Gamma \circ Ind$ has size $2^{\Omega(d-d_0)}$.*

Proof. Let $f_i' = f_i \circ \mathrm{Ind}$ and $h_i' = h_i \circ \mathrm{Ind}$. Then $\Gamma \circ \mathrm{Ind} = \{f_1' = 0, \ldots, f_m' = 0; h_1' \geq 0, \ldots, h_s' \geq 0\}$. The input variables to $\Gamma \circ \mathrm{Ind}$ are $\{w_{i0}, w_{i1}, w_{i2} | i \in [n]\}$, where $x_i = \mathrm{Ind}(w_{i0}, w_{i1}, w_{i2})$. We refer to w_{i0} as the selector variable of the indexing gadget, while w_{i1} and w_{i2} are termed as data variables.

Now, towards a contradiction, assume that we have an $\mathsf{SOS}_{\pm 1}$ refutation

$$\pi = (q_0, \ldots, q_s; p_1, \ldots, p_m)$$

of $\Gamma \circ \mathrm{Ind}$ with a size of $2^{c(d-d_0)}$ for a sufficiently small $c \in (0,1)$:

$$q_0 + \sum_{i=0}^{s} q_i h'_i + \sum_{i=1}^{m} p_i f'_i = -1.$$

Consider an assignment ρ_i to the selector variable w_{i0}, setting the gadget to one of the two data variables w_{ij}, where $j \in \{1,2\}$. We denote the other data variable as irrelevant. (If ρ_i sets $w_{i0} = -1$, then w_{i2} is irrelevant; otherwise, w_{i1} is irrelevant.) For an assignment ρ to selector variables and a monomial t over w_{ij} variables, we deem it irrelevant w.r.t ρ if it contains an irrelevant data variable. Moreover, we term a monomial as fat if it contains more than $d - d_0$ data variables.

Considering a uniformly random assignment ρ to selector variables, note that a fat monomial becomes irrelevant w.r.t. ρ with a probability of at least $1 - 1/2^{d-d_0}$. Let H be the set of fat monomials among the polynomials $(q_0, \ldots, q_s; p_1, \ldots, p_m)$. Since $|H| \le 2^{c(d-d_0)}$, by the union bound, there exists an assignment ρ to selector variables such that every monomial in H is irrelevant w.r.t ρ. We select such a restriction ρ.

Now, we observe:

- $\Gamma \circ \mathrm{Ind}|_\rho$ reduces to Γ over relevant data variables.
- For each $i \in [n]$, let w_{ij_i}, $j \in \{1,2\}$, be irrelevant data variables under assignment ρ. Then, π under restriction ρ becomes:

  $$q_0|_\rho + \sum_{i=0}^{s} (q_i|_\rho)(h'_i|_\rho) + \sum_{i=1}^{m} (p_i|_\rho)(f'_i|_\rho) = -1.$$

  Since $h'_i|_\rho = h_i$ and $f'_i|_\rho = f_i$, and since a sum-of-squares (the polynomials q_i's) restricted by ρ is still a sum-of-squares, we see that $\pi|_\rho$ is an $\mathsf{SOS}_{\pm 1}$ refutation of Γ.

Note that the restrictions $q_i|_\rho$, $p_i|_\rho$, may still contain irrelevant data variables, which eventually cancel out in the refutation $\pi|_\rho$. We eliminate these by assigning them values in $\{+1,-1\}$ uniformly at random and considering the expected resulting value on each side of the equation. Since the refutation $\pi|_\rho$ is a polynomial identity, it will remain an equality if we take expectations on both sides.

Letting E_I denote $\mathsf{E}_{w_{1j_1},\ldots,w_{nj_n}}$, and using linearity of expectation, we get

$$\begin{aligned} -1 &= \mathsf{E}_I \left[q_0|_\rho + \sum_{i=0}^{s} (q_i|_\rho) h_i + \sum_{i=1}^{m} (p_i|_\rho) f_i \right] \\ &= \mathsf{E}_I \left[q_0|_\rho \right] + \sum_{i=0}^{s} \mathsf{E}_I \left[q_i|_\rho \right] h_i + \sum_{i=1}^{m} \mathsf{E}_I \left[p_i|_\rho \right] f_i. \end{aligned}$$

Note that if a polynomial is a sum of squares of polynomials over variables taking values in $\{+1,-1\}$, then assigning a subset of variables to values uniformly and randomly results in a random polynomial whose expectation is still a sum of squares. For instance, with a single square $Q = P^2 = (sx+r)^2$ where x takes values in $\{+1,-1\}$ and s, r are x-free, $\mathsf{E}_x[Q] = \mathsf{E}_x[(sx+r)^2] = \mathsf{E}_x[s^2x^2 + r^2 + 2xsr] = s^2 + r^2$. Hence we see that

$$\pi' = (\mathsf{E}_I[q_0|_\rho], \mathsf{E}_I[q_1|_\rho], \ldots, \mathsf{E}_I[q_s|_\rho]; \mathsf{E}_I[p_1|_\rho], \ldots, \mathsf{E}_I[p_m|_\rho])$$

is a valid $\mathsf{SOS}_{\pm 1}$ refutation of Γ.

Furtherrmore, since under ρ each fat monomial in π contains an irrelevant variable, all the fat monomials vanish under expectation. Thus, each polynomial in π' has degree less than $d - d_0$, and π' is a refutation of Γ of degree less than d, leading to a contradiction. ◀

This improves Sokolov's lifting theorem for $\mathsf{SOS}_{\pm 1}$, where he lifted $\mathsf{SOS}_{\pm 1}$ degree d to $\exp(\Omega(\frac{(d-d_0)^2}{n}))$ $\mathsf{SOS}_{\pm 1}$ size, where d_0 represents the degree of the initial polynomial system. Consequently, his findings are only significant when $d = \omega(\sqrt{n})$, whereas our results are applicable for any superconstant degree lower bound. One notable example is the ordering principle; as shown in [15], its SOS degree is $\Omega(n^{1/4})$. Thus, lifting the ordering principle with a one-bit indexing gadget will yield exponential size lower bounds using our lifting result, whereas previous results would fail to achieve this.

4 PC with extension variables over finite fields

We now consider the setting where the encoding is over $\{0, 1\}$, the arithmetic is over finite fields, and extension variables are allowed; this is the setting for our second main result Theorem 1.2. In this setting, a size lower bound was obtained in [11] provided the extension variables are subquadratic in number and at most logarithmic in arity. We follow that approach but improve the result substantially. In this section, we first outline the framework of [11], then describe at a high level the outline of our proof of Theorem 1.2, and then present the relevant definitions/lemmas from [11] that we need to use. The actual formal proof of Theorem 1.2 appears in the next section.

4.1 The approach in [11]

We first outline the framework of [11], whose lower bounds we improve. The proof of the lower bound in [11] proceeds as follows.

Given a small refutation of a well chosen tautology F in PC with extension variables, pick an extension variable z with extension axiom $z - Q$ that contributes to a lot of pairs of terms of high Quadratic degree (which is a notion similar to Quadratic degree for ± 1 variables as in Definition 3.2, but generalized to $\mathbb{F}_p$-valued variables; see Definition 4.5). Extension variables are not necessarily Boolean; z can take a subset of values in the underlying field (over all possible values to the Boolean variables in Q). If this subset includes zero, apply the partial assignment that sets $z = 0$ to the proof to remove all contributions of z to Quadratic degree.

If not, z appears in each line of the proof in the form $P_{\ell-1}z^{\ell-1} + \cdots + P_1 z + P_0$ where ℓ is the least value such that z^ℓ is a constant. The contributions of z to Quadratic degree therefore come from interactions of the polynomials $P_i z^i$ and $P_j z^j$, over all pairs (i, j), $i \neq j$. Now pick a good pair (i, j) which contributes at least a $1/p^2$ fraction of the contributions of z to high Quadratic degree. The key step is to obtain a proof which separates the pair of polynomials P_i and P_j in each line into two different lines, using an operation called Split, see Definition 4.12. (Again, this is similar in spirit to the Split operation from Definition 3.3, but more nuanced.) Split essentially equates each line P to a polynomial of the form $R_1 z^i + R_0 z^j$, and solves for R_1 and R_0 in terms of P. In order for Split to output a valid proof, though, some preconditions needs to be satisfied: the axioms need to be free of z except for the range axiom for z, and this range needs to be such that z^i and z^j are linearly independent, i.e. $z^i \neq cz^j$ or $z^{i-j} \neq c$. That is, z needs to take on at least two values a, b such that $a^{i-j} \neq b^{i-j}$. Therefore there are two tasks at hand: getting rid of the extension axiom $z - Q$, and doing it in such a way that z^{i-j} is not set to a constant. It is shown that a restriction to Q can be chosen that sets it to the form $(b - a)x + a$, with a, b satisfying the precondition for Split. Once this happens, Split is applied to reduce a fraction of high degree terms (after applying an additional restriction to make sure x does not occur in the axioms, and then setting $x = (b - a)^{-1}(z - a)$ in order to get rid of the extension axiom).

This process is repeated until the proof is of low Quadratic degree. Then an argument from [18] (adapting Lemma 3.7 to the extension-variables setting) is used to move to a low (usual) degree proof of the tautology $F_{|_\rho}$, where ρ is the union of all restrictions ρ_i applied in this process. This contradicts the degree lower bound for $F_{|_\rho}$. An important element of this proof is to ensure that none of the restrictions ρ_is make the tautology easy. To ensure this, cleanup operations are performed at each iterations using additional restrictions, to restore to a sub-copy of F where hardness is preserved. These operations work correctly provided each extension variable depends on only $O(\log n)$ of the n variables of F. Also the number of iterations of this process has to be bounded as well, which gives an upper bound of $o(n^2)$ on the number of extension variables in order to get mildly exponential lower bounds.

4.2 Our proof outline

We largely follow the above approach for our lower bound, but show that sub-quadratically many extension variables with extension axioms that are polynomial sized and depend on a polynomial fraction of original variables can be handled.

For this we first reduce the above problem to handling low degree extension variables, by a simple parity lift. Let F be a tautology and let F' be obtained by replacing each input variable x of F by a two bit parity gadget, i.e. $x = w_1 \oplus w_2$. Suppose that there is a PC proof of F' which uses extension variables of size bounded by N^c. In each copy of the gadget we select one variable at random and set the other variable to zero or one with equal probability, recovering a copy of F (possibly renaming some variables by their negations). It is easy to see using a probabilistic argument that there exists a restriction where each extension variable is of degree at most $O(\log N)$.

Thus the problem reduces to proving lower bounds for F where extension axioms $z - Q$ are degree bounded. We then show that when dealing with such extension variables, the aforementioned process that picks an extension variable z and either sets it to zero or restricts it to the form $(b-a)x + a$ can be performed with restrictions whose hamming weight is bounded by the degree of Q.

Finally, we observe that the tautology $\mathrm{PHP}_n^{m,r}$ introduced by Razborov [17], which maps all r sized subsets of $[m]$ to n holes, is immune to such restrictions and can be cleaned up to restore hardness. This completes our proof.

For the rest of the article, we fix the finite field $\mathbb{F}_p$, $p > 2$ (since for the case of $p = 2$ lower bounds for PC with extension variables can be obtained through standard size-degree tradeoffs, see the discussion in [11], Section 3).

4.3 Relevant material from [11]: Support, Quadratic Degree, Split

Here we introduce the terminology of [11] and state some lemmas about Quadratic degree and Split from the same. The reader familiar with [11] can directly jump to Section 5.

As noted above, the notions of Quadratic degree and Split here are not the same as in Definitions 3.2 and 3.3. To define them appropriately in this setting, we first need some auxiliary notions.

▶ **Definition 4.1** (Support of a variable, Singular/Nonsingular variables. [11], Defs 10,11). *Let $z - Q(w_{i_1}, \dots, w_{i_\kappa}) = 0$ be an extension axiom associated with z. The set $vars(Q)$ is defined as $vars(Q) = \{w_{i_1}, \dots, w_{i_\kappa}\}$, and is sometimes also written as $vars(z)$, the set of variables that z depends on.*

The support of z, $supp(z) \subseteq [0, p-1]$, is equal to the set of all values that z can take under Boolean assignments to $vars(z)$. That is, $supp(z) = \{Q(\alpha) \mid \alpha \in \{0,1\}^{|vars(Q)|}\} \subseteq [0, p-1]$. Sometimes this is also denoted by $supp(Q)$.

We say that z is a Singular variable if $0 \in supp(z)$, otherwise it is NonSingular.

For a Boolean variable w, $supp(w) = \{0,1\}$ as enforced by the Boolean axiom $w^2 = w$.

As we apply restrictions to a proof (and hence to all the defining axioms), 0 may get removed from the support of a variable. Thus an extension variable can change from Singular to NonSingular, but not the other way around. However, Boolean variables that are not set by the restriction are always Singular.

▶ **Definition 4.2** ([11], Definition 12). *Let $A \subseteq [1, \ldots, p-1]$, $A \neq \emptyset$. Define $\ell(A)$ to be the least $\ell \in [1, p-1]$ such that the set $\{a^\ell \mid a \in A\}$ is singleton. For a Nonsingular z, define $\ell(z) = \ell(supp(z))$.*

The following lemma from [11] is stated without proof.

▶ **Lemma 4.3** ([11], Lemma 13). *Let z be a Nonsingular extension variable with extension axiom $z - Q = 0$. Then the following polynomial equations are implied by (and therefore derivable from) the extension axiom for z plus the Boolean axioms for all variables in $vars(Q)$, in degree at most $|vars(Q)|$.*

1. *$z - Q' = 0$, where Q' is the multilinear version of Q;*
2. *For any $A' \subseteq [0, p-1]$ such that $supp(z) \subseteq A'$, $\Pi_{a \in A'}(z - a) = 0$;*
3. *$z^{\ell(z)} - c = 0$ for some $c \in \mathbb{F}_p^*$.*

In particular, if z is Nonsingular, then the polynomial equation $z^{p-1} - 1 = 0$ is implied by $z - Q = 0$ together with the Boolean axioms for vars(Q).

▶ **Definition 4.4** ([11], Definition 14). *For a term t and a variable w, $\deg(t, w)$ is equal to the degree of w in t. Note that since we are working over $\mathbb{F}_p$, $\deg(t, w) < p$ for any variable w. For a term t, the degree of t, denoted $\deg(t)$, equals $\sum_{w \in vars(t)} \deg(t, w)$.*

Quadratic degree

The following definition of Quadratic degree is taken from [11].

▶ **Definition 4.5** (Quadratic degree [11], Definition 10). *Let V be a set of variables and let S be a subset of V. For a pair of terms t_1, t_2 over V, and a variable $w \in V$, we define $Qdeg^S(t_1, t_2, w)$ as follows. If $w \in S$, then $Qdeg^S(t_1, t_2, w) = 1$ if w occurs in at least one of t_1 or t_2; if $w \notin S$, then $Qdeg^S(t_1, t_2, w) = 1$ if and only if $\deg(t_1, w) \neq \deg(t_2, w)$. The overall quadratic degree of the pair t_1, t_2, $Qdeg^S(t_1, t_2)$, is equal to $\sum_{w \in V} Qdeg^S(t_1, t_2, w)$. The quadratic degree of a polynomial P is equal to the maximum quadratic degree over all pairs (t_1, t_2) such that $t_1, t_2 \in P$. For a proof Π, the quadratic degree of Π is the maximum quadratic degree over all polynomials $P \in \Pi$.*

▶ Remark 4.6. The above definition of Quadratic degree treats the variables in the set S differently from the rest of the variables. Typically S will not be explicitly specified, but will be assumed to be the set of Singular variables. This means that the notion of Quadratic degree depends on knowing which variables have zero in their support. For instance, for the pair $(z_1, z_1 z_2)$, the Quadratic degree is two if z_1 can take the value zero and one if z_1 does not take the value zero. We observe that Quadratic degree always decreases when a variable changes from being Singular to Nonsingular, and make sure that this is the case when we prove our lower bound. This follows the approach of [11] whose lemmas we state below.

▶ **Lemma 4.7** ([11], Lemma 16). *Let V be a set of variables and let S and T be subsets of V such that $T \subseteq S$. Then for any two terms t_1, t_2 over V, $Qdeg^T(t_1, t_2) \leq Qdeg^S(t_1, t_2)$.*

Since applying a restriction cannot make NonSingular variables Singular, Lemma 4.7 implies that the Quadratic degree of any two terms t_1, t_2, $Qdeg(t_1, t_2)$ with respect to the currently Singular variables cannot increase after applying the restriction. This is stated as Corollary 17 in [11].

▶ **Lemma 4.8** ([11], Lemma 20). *Let Π be a $\mathsf{PC} + \mathsf{Ext}$ refutation of F and let z be a Nonsingular variable. Let Π' be the proof obtained from Π by reducing each line of Π by $z^{\ell(z)} - c = 0$ for some $c \in \mathbb{F}_p^*$. Then for any $d \geq 0$, the number of pairs of terms of Quadratic degree at least d in Π' is at most that of Π.*

We will use the following lemma from [11], which is a generalization of the argument from [18] that shows how to convert a proof with low Quadratic degree to one with low degree.

▶ **Lemma 4.9** ([11], Lemma 21). *Let F be a set of unsatisfiable polynomials of degree d_0 with a PC refutation of Quadratic degree at most $d \geq d_0$ over $\mathbb{F}_p$. Then F has a PC refutation of degree at most $3pd$.*

The Split operation

In this section, we define the operation Split and state its properties. We will only need to handle variables whose only axiom is $(z - a)(z - b) = 0$ for $a, b \in \mathbb{F}_p^*$, as we will apply an assignment to any general extension variable to reduce to this case. Below we state the relevant lemmas from [11].

▶ **Lemma 4.10** ([11], Lemma 23). *Let z be an extension variable such that $supp(z) = \{a, b\}$, where $a \neq b$ and $a, b \in \mathbb{F}_p^*$ and let P be any polynomial. Then, for any two distinct numbers i, j where $i < j$ and $a^{j-i} \neq b^{j-i}$, there exists a unique polynomial $R = R_0 z^i + R_1 z^j$ such that $R = P \mod (z - a)(z - b)$.*

▶ Remark 4.11. It can be checked that for polynomial $P = \sum_{l<\ell(z)} P_l z^l$, the polynomials R_0, R_1 have the following form:

$$R_0 = P_i + \sum_{l<\ell(z), l\neq i,j} c_{0l} P_l$$

$$R_1 = P_j + \sum_{l<\ell(z), l\neq i,j} c_{1l} P_l$$

for some constants $c_{1i}, c_{0i} \in \mathbb{F}_p$. Note that any pair of terms (t_1, t_2) occurring in either R_1 or R_0 also occurs in P as $(t_1 z^{i'}, t_2 z^{j'})$ with $(i', j') \neq (i, j)$. That is, the contribution to Quadratic degree of P by the interaction of z^i and z^j is removed.

▶ **Definition 4.12** (Split [11], Definition 24). *Let z be an extension variable with extension axiom $z - Q = 0$ such that $supp(z) = \{a, b\} \subseteq [1, \ldots, p-1]$. For any polynomial P and for every $i < j$ such that $a^{j-i} \neq b^{j-i}$, let $R = R_0 z^i + R_1 z^j$ be the unique polynomial given by Lemma 4.10 such that $R = P \mod (z - a)(z - b)$. Then $Split_{z,i,j}(P)$ is defined to be the pair of polynomials $\{R_0, R_1\}$. For a proof Π, and an extension variable z such that $supp(z) = \{a, b\}$, $Split_{z,i,j}(\Pi)$ is the sequence of lines $Split_{z,i,j}(P)$, over all $P \in \Pi$.*

▶ **Lemma 4.13** ([11], Lemma 25). *Let Π be a refutation of a set of unsatisfiable polynomials F. Let z be a variable that occurs in Π such that the polynomials in F do not contain z except for the axiom $(z-a)(z-b)=0$ for some $a,b \in \mathbb{F}_p^*$. Then for any i,j such that $i<j$ and $a^{j-i} \neq b^{j-i}$, $\Pi' = Split_{z,i,j}(\Pi)$ forms a valid refutation of F modulo $(z-a)(z-b)$.*

5 Proof of the lower bound from Theorem 1.2

5.1 The tautology

We use the $\mathrm{PHP}_n^{m,r}$ tautology defined in [17]; it is a variant of the Pigeonhole principle. In this variant, there are m "fractional" pigeons, r fractional parts add up to a whole "pigeon", and there are n holes. The Boolean variables determine which part goes into which hole. A fractional part can participate in multiple pigeons, and can be assigned to multiple holes. The constraints enforce that no two r-sized subsets are mapped to the same hole, and no r-sized subset is mapped to more than one hole. When $\binom{m}{r} > n$, this is unsatisfiable. We describe the formula formally below.

▶ **Definition 5.1** ($\mathrm{PHP}_n^{m,r}$; Def 4.1 in [17]). *Let $m,n,r>0$ be such that $\binom{m}{r} > n$. Let x_{ij}, for $i \in [m]$, $j \in [n]$, be variables that indicate the mapping of elements of $[m]$ to holes in $[n]$. For a subset I of $[m]$, abbreviate the term $\prod_{i\in I} x_{ij}$ to t_{Ij}; note that t_{Ij} is only shorthand and not a variable in the formula. Then $PHP_n^{m,r}$ is the following set of equations.*

$$\begin{aligned}
&t_{I1}+t_{I2}+\cdots+t_{In}=1 && \forall I\subset[m], |I|=r\\
&t_{Ij}=0 && \forall I\subset[m], |I|=r+1; \forall j\in[n]\\
&t_{Ij_1}t_{Ij_2}=0 && \forall I\subset[m], |I|=r; \forall j_1,j_2\in[n], j_1\neq j_2.
\end{aligned}$$

(Note that the last set of constraints is already implied by the first two constraint sets. It is nonetheless included, in [17], where a degree lower bound is shown even when these constraints are explicitly given and do not have to be derived.)

Additionally, we note that although the axioms defining $\mathrm{PHP}_n^{m,r}$ do not have small CNF representations, they can be represented by linear sized depth two $\mathsf{AC}^0[p]$ circuits. For our size lower bound on $\mathsf{PC}+\mathsf{Ext}$, we use an XOR lifted version of this tautology, which still has linear sized depth four $\mathsf{AC}^0[p]$ circuits. Therefore, we reiterate that our lower bounds still imply lower bounds for some fragment of $\mathsf{AC}^0[p]$-Frege.

We state the lower bound from [17] on the degree of PC proofs for $\mathrm{PHP}_n^{m,r}$.

▶ **Proposition 5.2** ([17], Theorem 4.2). *For any ground field $\mathbb{F}$ and any $m,r,n>0$ such that $\binom{m}{r} > n$, $PHP_n^{m,r}$ requires proofs of degree $n/2+1$ to refute in PC over $\mathbb{F}$.*

5.2 The lower bound

We begin by showing in Theorem 5.5 a weak size lower bound for the $\mathrm{PHP}_n^{m,r}$ formulas in PC when extension variables are allowed, provided the degree of the extension axioms is bounded. To establish the strong lower bound in Theorem 1.2, as discussed in Section 4.2, we show that a lift with the parity gadget (an XOR-ification of the formula), followed by a well-chosen restriction, achieves a degree-reduction of the extension axioms, and use Theorem 5.5.

The weak size lower bound of Theorem 5.5 also uses degree reduction, but it reduces the quadratic degree of the proof. A crucial ingredient in the quadratic-degree-reduction step is finding low-Hamming-weight assignments with certain nice properties. We first prove the existence of these assignments, and then show the weak lower bound.

▶ **Lemma 5.3.** *Let z be an extension variable with the extension axiom $z - Q$ and let $l < p$ be a constant such that Q^l is not a constant, i.e. supp(Q^l) is not singleton. Then there exists a partial assignment σ of Hamming weight at most $l \deg(Q)$, such that for some $x \in vars(z)$, $Q_{|_\sigma} = (b - a)x + a$ for some a, b with $a^l \neq b^l$.*

Proof. Let X be the set of variables of Q and let x be a variable that appears in Q^l. Since Q^l is not a constant, such a variable always exists. Fix a total ordering over monomials in X that respects degree. Let $\mathcal{M}$ be the least monomial in Q^l according to this ordering that contains x. Let σ be an assignment to $X \setminus \{x\}$ obtained as follows: we set every variable in $\mathcal{M}$ other than x to one, and every other variable in $X \setminus \{x\}$ to zero. Note that this sets every monomial lesser than $\mathcal{M}$ to a constant since it does not contain x. The same is true for monomials greater than $\mathcal{M}$ that do not contain x. Since the ordering respects degree and $\mathcal{M}$ is minimal according to it, any monomial greater than $\mathcal{M}$ that contains x also contains at least one variable that is not in $\mathcal{M}$ and hence is set to zero by σ. Therefore, $\mathcal{M}$ is the only monomial containing x in Q^l that survives the restriction under σ; thus $(Q^l)_{|_\sigma} = \alpha x + \beta$ for some $\alpha \neq 0$. Since $(Q^l)_{|_\sigma} = (Q_{|_\sigma})^l$ is not a constant and σ sets all variables except x, $Q_{|_\sigma}$ cannot be a constant and must take the form $(b - a)x + a$, where $a^l = \beta \neq \alpha + \beta = b^l$. By our choice of σ, it has Hamming weight at most the degree of the monomial $\mathcal{M}$, which is bounded above by $\deg(Q^l) \leq l \deg(Q)$. ◀

This lemma will enable us to satisfy the necessary precondition for applying a Split operation on the variable z, when required.

▶ **Corollary 5.4.** *Suppose that z is an extension variable with the extension axiom $z - Q$, where $0 \in$ supp(Q). Then there exists an assignment of Hamming weight at most $p \deg(Q)$ which sets Q to zero.*

Proof. If supp(Q) $= \{0\}$, then any assignment to vars(Q) will do.

If supp(Q) is not a singleton, then for every $l \in [p-1]$, Q^l is not a constant, and thus we can choose $l = p - 1$ in Lemma 5.3. We thus obtain a partial assignment σ of Hamming weight $(p-1)\deg(Q)$ such that $Q_{|_\sigma} = (b - a)x + a$, with $(Q_{|_\sigma})^{p-1}$ not a constant; i.e. $a^{p-1} \neq b^{p-1}$. This means that exactly one of a, b is zero. Setting the value σ_x to be 0 if $a = 0$ and 1 otherwise, we see that $Q_{|_{\sigma \cup \sigma_x}} = 0$. The Hamming weight of $\sigma \cup \sigma_x$ is at most $(p-1)\deg(Q) + 1 \leq p \deg(Q)$. ◀

We now state and prove our main theorem of this section.

▶ **Theorem 5.5.** *Let Π be a PC refutation of $PHP^{m,r}_n$ with M extension variables, each of degree $\leq k$ and depending on $\leq \kappa$ variables of $PHP^{m,r}_n$, such that $r > 2pk$. Then the size of Π is at least $\exp(\Omega(n^2/(M + mn)\kappa k))$.*

Proof. Let s be the size of the given refutation Π.

For a threshold d that we will choose later and depends on s, we will first show how to reduce the Quadratic degree to at most d. This will be achieved by finding a suitable restriction, in stages, that kills all quadratic terms of quadratic degree more that d. In the process, the restricted formula will become $\mathrm{PHP}^{m,r}_{n'}$ for some $n' \in \Theta(n)$. Using Lemma 4.9 we will convert this to a PC proof of $\mathrm{PHP}^{m,r}_{n'}$ of degree at most $3pd$ but with extension variables, and then by directly substituting the extension axioms, to a PC proof of degree at most $3pkd$ without extension variables. Finally, using the degree lower bound from Proposition 5.2, we will obtain the desired lower bound on s.

(Note that the notion of quadratic degree is defined with respect to some set S of variables. We assume that S is the set of Singular variables (those which can potentially take the value 0), and while finding the suitable restriction, we update S in each stage. Boolean variables are Singular unless set to 1 by the restriction.)

Let H be the set of all pairs of terms in Π of Quadratic degree more than d. We know that $|H| \leq s^2$. In each iteration, we will find a restriction that removes a fraction α of the pairs from H, for $\alpha = \frac{d}{4p^2(M+mn)}$, and removes no more than κ holes from the formula. Thus for a t satisfying $(1-\alpha)^t|H| \leq (1-\alpha)^t s^2 < 1$, after t iterations, no high-degree quadratic terms survive, and the number of remaining holes is $n' \geq n - t\kappa = n(1-t\kappa/n)$. Since $1-\alpha \leq e^{-\alpha}$, note that t is roughly $2\log s/\alpha$. At this point, the choice for d is clearer; we choose d so that $t\kappa/n$ is a small enough constant; say $t\kappa/n \leq 1/2$. Choosing d so that $\alpha = \frac{4\kappa \log s}{n}$ does the trick; in particular, $d = \frac{16p^2(M+mn)\kappa \log s}{n}$. With this choice of d, continuing with the outline above, we obtain a PC proof of degree $3pkd$ without extension variables for $\mathrm{PHP}_{n'}^{m,r}$ with $n' \geq n/2$. From Proposition 5.2, we conclude that $3pkd \geq n'/2 \geq n/4$, and plugging in the chosen value of d, we see that $\log s \geq \Omega\left(\frac{n^2}{kp^3(M+mn)\kappa}\right)$.

Now we come to the main part of the proof, namely showing how to obtain the desired restriction in each iteration.

In each iteration, we first perform the following preprocessing steps. For each extension variable z with extension axiom $z - Q$, we compute its support and check whether zero is in it. If not, we compute $\ell(z)$ (Definition 4.2) and reduce the proof by $z^{\ell(z)} = c$. By Lemma 4.3 the latter is derivable from the extension axiom, and by Lemma 4.8, it does not raise the size of H. Moreover, our measure of Quadratic degree can only decrease when variables switch from Singular to Nonsingular; see comment after Lemma 4.7.

We then pick a variable y that by an averaging argument contributes to the quadratic degree of at least a $d/(M+mn)$ fraction of pairs in H. There are three cases to consider.

Case 1. y is an original Boolean variable, say x_{uv} for some $u \in [m]$, $v \in [n]$. We choose the restriction that sets all variables $x_{w,v}$ to 0, thus removing the hole v from the formula. Since x_{uv} is also set to 0 this way, $d/(M+mn)$ fraction of pairs in H are killed.

Case 2. y is an extension variable, say z, with the extension axiom $z - Q$, with $0 \in \mathrm{supp}(Q)$, with $\deg(Q) \leq k$ and $|\mathrm{vars}(Q)| \leq \kappa$.

By Corollary 5.4, we can find an assignment σ to $\mathrm{vars}(Q)$ that has Hamming weight at most pk and sets Q to 0. We apply this assignment to the proof, additionally setting z to zero in the proof as well. We then look at how this assignment affects the tautology, and apply an additional assignment to restore to $\mathrm{PHP}_{n'}^{m,r}$ where $n' \geq n - \kappa$.

We say that a hole v is affected if for some u, the variable x_{uv} is set by σ. Note that at most κ holes are affected since σ only sets variables in $\mathrm{vars}(Q)$. We say that an assignment commits a pigeon $I \in \binom{[m]}{r}$ to a hole v if it sets the term t_{Iv} to 1. Now note that σ does not commit any pigeon I to any affected hole v, because each term t_{Iv} is the product of r variables, and the Hamming weight of σ is at most pk which is less than $r/2$. Thus we are free to remove an affected hole from the formula. We do so by setting to zero all unset variables x_{uv} for each affected hole v; this makes $t_{Iv} = 0$ for all affected holes v and all pigeons I. The resulting formula is $\mathrm{PHP}_{n'}^{m,r}$ where $n' \geq n - \kappa$, and applying the restriction to the current refutation gives a new refutation of this reduced formula with at most $(1 - d/(M+mn))|H|$ pairs of Quadratic degree d or more.

Case 3. y is an extension variable, say z, with the extension axiom $z - Q$, and $0 \notin \mathrm{supp}(Q)$. This is the trickiest case.

We need to find a suitable assignment to the variables in $\mathrm{vars}(Q)$ to kill many high-degree quadratic terms involving z. Recall from Definition 4.5 that, since z is non-singular, z contributes to the degree of a quadratic term pair through terms t_1, t_2 where the degrees of z in t_1 and t_2 are different. By averaging, we can pick indices $0 \leq i < j \leq p-1$ such that pairs of terms of the form $(t_1 z^i, t_2 z^j)$ contribute at least a $d/p^2(M+mn)$ fraction of the contribution of z. Since we had preprocessed using Lemmas 4.3 and 4.8, we know that

$i, j < \ell(z)$, so z^{j-i} is not a constant. Hence, using Lemma 5.3 with $l = j - i$, once again we obtain an assignment σ of Hamming weight $\leq lk$ such that $Q_{|_\sigma} = (b-a)x_{uv} + a$, where $a^{j-i} \neq b^{j-i}$ and x_{uv} is some variable of $\mathrm{PHP}_n^{m,r}$.

We would like to apply $\mathrm{Split}_{z,i,j}$ to remove the contribution of these term pairs with z^i, z^j (see Section 4.3 and Remark 4.11) and reduce high Quadratic degree terms. But first we need to meet the preconditions for applying $\mathrm{Split}_{z,i,j}$. In particular, we need to get rid of all axioms containing z, except for $(z-a)(z-b) = 0$; even the extension axiom $z = (b-a)x_{uv} + a$ must be eliminated. We also need to restore to a version $\mathrm{PHP}_{n'}^{m,r}$. To this end, we apply σ and perform cleanup in a way similar to Case 2, before applying $\mathrm{Split}_{z,i,j}$. The only difference is that here we need to get rid of all axioms containing x_{uv}, without actually setting the latter.

We handle holes v' other than v affected by σ exactly as in Case 2; all variables touching this hole ($x_{wv'}$) but unset by σ are now set to 0, eliminating hole v'.

For the hole v, all variables touching this hole (x_{wv}) but unset by σ are now set to 0, but x_{uv} is left unset. Nonetheless, we claim that all occurrences of x_{uv} in the axioms are now eliminated. This is because every such occurrence is in a term t_{Iv}, for some subset I of $[m]$ of size at least r. Since σ sets at most $lk < pk$ variables to one and $r > 2pk$, each such occurrence contains at least two variables unset by σ, and in particular contains an unset variable other than x_{uv}. Therefore, setting all variables of hole v (other than x_{uv}) which are unset by σ to zero gets rid of all such occurrences and thereby eliminates hole v.

Thus, we end up with a refutation of $\mathrm{PHP}_{n'}^{m,r}$ for $n' \geq n - \kappa$ such that all affected holes are eliminated, and x_{uv} is still unset but does not appear in the axioms.

We now intend to substitute $x_{uv} = (b-a)^{-1}(z-a)$. Note that under this substitution, the extension axiom gets eliminated (becomes $0 = 0$) and the Boolean axiom $x_{uv}^2 - x_{uv} = 0$ reduces to $(z-a)(z-b) = 0$. This is possible by Lemma 4.3(2), and will enable us to satisfy all the preconditions to apply Split on z. However, there is a still a catch. The substitution might actually blow up the number of pairs in H, because it creates three additional pair of terms for every pair of terms (t_1, t_2) containing x. To handle this, we note that if the substitution blows up the number of high Quadratic degree pairs to more than $3d|H|/4p^2(M+mn)$, then this implies that at least a $d/4p^2(M+mn)$ fraction of pairs of terms in H must have contained x_{uv} before this substitution. (The same argument is also used in [11].) In this case, we can just set $x_{uv} = 0$ instead of the above substitution; this will remove a $d/4p^2(M+mn)$ fraction of pairs of terms (and prevent the need to use Split). Otherwise, we apply the substitution, introducing at most $3d|H|/4p^2(M+mn)$ new pairs of terms, and then use $\mathrm{Split}_{z,i,j}$ to obtain a valid refutation of the reduced formula (Lemma 4.13), removing at least the $d|H|/p^2(M+mn)$ pairs of terms which had quadratic degree with a contribution from z, i, j. Either way, the number of high-degree quadratic terms reduces by a fraction at least $d/4p^2(M+mn)$. Thus, we obtain a refutation of $\mathrm{PHP}_{n'}^{m,r}$ with at most $(1 - d/4p^2(M+mn))|H|$ pairs of terms of Quadratic degree at least d, with $n' \geq n - \kappa$.

This completes the description of how to extract a good restriction in each iteration. The fraction of high-degree Quadratic pairs eliminated is at least $d/(M+mn)$ in the first two cases and at least $d/4p^2(M+mn)$ in case 3. So in every case, at least $\alpha = d/4p^2(M+mn)$ fraction of the pairs is removed. With the analysis given in the beginning of this proof, the proof of Theorem 5.5 is now complete. ◀

Finally, applying a lift with the XOR_2 gadget and by choosing the parameters carefully, we obtain our claimed lower bound of Theorem 1.2.

▶ **Theorem 1.2.** *For every $N > 0$ large enough, any $1 > \epsilon, \delta > 0$, constant $c > 0$, and prime p, there exists a tautology F over N variables such that any PC refutation of F over $\mathbb{F}_p$ with $N^{1+\epsilon(1-\delta)}$ extension variables, each depending on $N^{1-\epsilon}$ variables of F and of size at most N^c, requires size* $\exp(\Omega(N^{\epsilon\delta}/\mathsf{poly}\log N))$.

Proof. Pick an arbitrary n, and set $r = 100p(c + 2)\log n$ and $m = 2r$, so that $\binom{m}{r} > n$. Let G be the formula $\mathrm{PHP}_n^{m,r}$. Let F be the formula obtained be composing G with the parity gadget on two variables; $F = G \circ \mathrm{XOR}_2$. That is, replace each variable x in G by the XOR of two new variables x^1 and x^2. The number of variables in the formula F is $N = 2mn = \Theta(n \log n)$.

Suppose we are given a PC refutation Π of F of size s, that uses no more than $N^{1+\epsilon(1-\delta)}$ extension variables, each of arity bounded by $N^{1-\epsilon}$ and size bounded by N^c. We will recover from Π a refutation of G, and then use Theorem 5.5 to obtain the stated lower bound on s.

Set $k = 10(c+2)\log N$; then $r > 2pk$. We will find a restriction that reduces F to G, and reduces the degree of all extension axioms in Π to at most k. Note that the total size of all the extension axioms put together is at most $N^c \times N^{1+\epsilon(1-\delta)} < N^{c+2}$. Let ρ be a restriction that independently, for each variable x of G, picks one of x^1, x^2 uniformly at random, and sets it to 0 or 1 with equal probability. For any term t of degree at least k, the probability that t survives after applying ρ is at most $(3/4)^k$. By the union bound, the probability that some term in an extension axiom survives ρ is at most $N^{c+2}(3/4)^k$, which is strictly less than 1 for our choice of k. Hence there exists a restriction ρ that sets exactly one variable in each XOR gadget, and which reduces all extension axioms to degree at most k. A suitable renaming of the surviving variables (and interchanging with the negated literal if necessary) recovers G.

We thus have a PC refutation $\Pi' = \Pi_{|_\rho}$ of G of size at most s. The number of extension variables in Π' is $M = N^{1+\epsilon(1-\delta)} = \tilde{O}(n^{1+\epsilon(1-\delta)})$, and each has arity at most $\kappa = \tilde{O}(n^{1-\epsilon})$ and degree at most $k = \Theta(\log n)$. Further, $M + mn = \theta(M)$. Also as already noted, $r > 2pk$. Hence by Theorem 5.5 we conclude that s is at least $\exp(\tilde{\Omega}(n^2/(M + mn)\kappa k)) = \exp(\tilde{\Omega}(n^{1+\epsilon}/n^{1+\epsilon(1-\delta)})) = \exp(\tilde{\Omega}(N^{\epsilon\delta}))$ (where the $\tilde{O}, \tilde{\Omega}$ notation hides $\mathsf{poly}\log$ factors). This is the claimed bound. ◀

6 Conclusion

We have obtained new size lower bounds for the PC proof system in two different settings.

Over the $\{\pm 1\}$ basis over any field, our lower bound is established through degree-to-size lifting (Theorem 1.1), and our method also yields a stronger (than previously known) degree-to-size lifting for the SOS proof system (Theorem 3.10).

Over finite fields of prime order, when extension variables are allowed, our lower bound is achieved by using XOR-ification on a carefully chosen unsatisfiable system of inequalities. We believe that this bound should also hold in fields of finite characteristic.

As mentioned in the precursor to our work [11], our generalization of PC can be viewed as a fragment of $\mathsf{AC}^0[p]$-Frege of depth "2.5", consisting of circuits with a mod p gate at the top, AND/OR gates in the middle, and a layer of gates that encode extension variables at the bottom. Thus lower bounds in this paper and those in [11] can be seen as progress towards proving lower bounds for depth 3 $\mathsf{AC}^0[p]$-Frege. However the latter are proved for CNF tautologies. Our bounds, while not for CNFs, are for tautologies that have small depth 2 $\mathsf{AC}^0[p]$ circuits and therefore can still be seen a progress towards this goal. It is an open problem to match our lower bounds for a CNF tautology.

References

1 Michael Alekhnovich and Alexander A Razborov. Lower bounds for polynomial calculus: Non-binomial case. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 190–199. IEEE, 2001.

2 Yaroslav Alekseev. A lower bound for Polynomial Calculus with extension rule. In Valentine Kabanets, editor, *36th Computational Complexity Conference, CCC 2021, July 20-23, 2021, Toronto, Ontario, Canada (Virtual Conference)*, volume 200 of *LIPIcs*, pages 21:1–21:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CCC.2021.21.

3 Christoph Berkholz. The relation between Polynomial Calculus, Sherali-Adams, and Sum-of-Squares proofs. In *35th Symposium on Theoretical Aspects of Computer Science (STACS 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

4 Sam Buss, Dima Grigoriev, Russell Impagliazzo, and Toniann Pitassi. Linear gaps between degrees for the Polynomial Calculus modulo distinct primes. *Journal of Computer and System Sciences*, 62(2):267–289, 2001.

5 Matthew Clegg, Jeffery Edmonds, and Russell Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 174–183, 1996.

6 Stephen A Cook and Robert A Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, 1979.

7 Nicola Galesi and Massimo Lauria. Optimality of size-degree tradeoffs for Polynomial Calculus. *ACM Transactions on Computational Logic (TOCL)*, 12(1):1–22, 2010.

8 Dima Grigoriev and Edward A Hirsch. Algebraic proof systems over formulas. *Theoretical Computer Science*, 303(1):83–102, 2003.

9 Dima Grigoriev, Edward A Hirsch, and Dmitrii V Pasechnik. Complexity of semi-algebraic proofs. In *STACS 2002: 19th Annual Symposium on Theoretical Aspects of Computer Science Antibes-Juan les Pins, France, March 14–16, 2002 Proceedings 19*, pages 419–430. Springer, 2002.

10 Russell Impagliazzo, Sasank Mouli, and Toniann Pitassi. The surprising power of constant depth algebraic proofs. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 591–603, 2020.

11 Russell Impagliazzo, Sasank Mouli, and Toniann Pitassi. Lower bounds for Polynomial Calculus with extension variables over finite fields. In *38th Computational Complexity Conference (CCC 2023)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

12 Russell Impagliazzo, Pavel Pudlák, and Jiri Sgall. Lower bounds for the Polynomial Calculus and the Gröbner basis algorithm. *Computational Complexity*, 8:127–144, 1999.

13 Matthias Krause and Pavel Pudlák. On the computational power of depth 2 circuits with threshold and modulo gates. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 48–57, 1994.

14 Mladen Miksa and Jakob Nordström. A generalized method for proving Polynomial Calculus degree lower bounds. In *30th Conference on Computational Complexity, CCC 2015*, volume 33 of *LIPIcs*, pages 467–487, 2015.

15 Aaron Potechin. Sum of squares bounds for the ordering principle. In *35th Computational Complexity Conference*, 2020.

16 Ran Raz and Iddo Tzameret. Resolution over linear equations and multilinear proofs. *Annals of Pure and Applied Logic*, 155(3):194–224, 2008.

17 Alexander A Razborov. Lower bounds for the Polynomial Calculus. *Computational Complexity*, 7:291–324, 1998.

18 Dmitry Sokolov. (Semi)Algebraic proofs over $\{\pm 1\}$ variables. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 78–90, 2020.

On the Relative Efficiency of Dynamic and Static Top-Down Compilation to Decision-DNNF

Alexis de Colnet
Algorithms and Complexity Group, TU Wien, Austria

Abstract

Top-down compilers of CNF formulas to circuits in decision-DNNF (Decomposable Negation Normal Form) have proved to be useful for model counting. These compilers rely on a common set of techniques including DPLL-style exploration of the set of models, caching of residual formulas, and connected components detection. Differences between compilers lie in the variable selection heuristics and in the additional processing techniques they may use. We investigate, from a theoretical perspective, the ability of top-down compilation algorithms to find small decision-DNNF circuits for two different variable selection strategies. Both strategies are guided by a graph of the CNF formula and are inspired by what is done in practice. The first uses a dynamic graph-partitioning approach while the second works with a static tree decomposition. We show that the dynamic approach performs significantly better than the static approach for some formulas, and that the opposite also holds for other formulas. Our lower bounds are proved despite loose settings where the compilation algorithm is only forced to follow its designed variable selection strategy and where everything else, including the many opportunities for tie-breaking, can be handled non-deterministically.

2012 ACM Subject Classification Theory of computation → Dynamic programming

Keywords and phrases Knowledge compilation, top-down compilation, decision-DNNF Circuits

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.11

Funding This work has been supported by the Austrian Science Fund (FWF), ESPRIT project FWF ESP 235.

1 Introduction

The foundation of *knowledge compilation* is the idea that different representations of a function facilitate solving different kind of problems. Classes of representations where specific problems become tractable are studied under the name of *(compilation) languages* [10, 11, 12]. The purpose of (knowledge) *compilers* is to transform (or compile) propositional formulas, circuits, or other, into a target language where some intractable problems become solvable in polynomial time. Compilers to languages where model counting is linear-time are, for obvious reasons, particularly investigated. In practice, one of the main language for doing model counting is *decision-DNNF*: the class of circuits in *decision decomposable negation normal form* [20], with compilers building decision-DNNF circuits from CNF formulas such as `c2d` [9], `dsharp` [19], `d4` [17] and the compiler version of `sharpsat-TD` [16, 15].

From a complexity theory point of view, compilation is often seen as a preprocessing task where only the size of the compiled form (that is, the output of the compiler) matters [4, 6]. In practice of course, running time matters, and perhaps is the priority since a compiler that ends rapidly is also guaranteed to construct a reasonably small compiled form (though the converse is not true). So practical compilers use strategies, heuristics and whatnot to try to terminate within a certain time window. But then we ask what are the consequences of these implementation choices on their ability to find small compiled forms. Are there formulas that admit small decision-DNNF circuits but for which our compilers always construct large circuits? In this paper, we answer this question positively for two, hopefully realistic, algorithms for top-down compilers of CNF to decision-DNNF. More precisely, we confront the two algorithms by finding formulas that are hard to compile for one, in the sense that it returns super-polynomial-size compiled forms, but easy to compile for the other, and vice versa. The two algorithms share many similarities and are *top-down*.

Top-down compilation refers to a compilation based on an exhaustive DPLL search that uses caching and independent components identification. Here, "exhaustive" means that the DPLL procedure does not stop after finding one model of the formula but keeps searching for all of them. Caching allows to save time and memory by preventing the compiler to work twice on the same formula [1, 23, 9]. Independent components identification also speeds up compilation by determining when a formula can be split into independent subformulas that are then compiled separately [14, 8, 1]. Without this mechanism, the compiled form are FBDDs (free binary decision diagrams) [13], which are generally significantly less succinct than decision-DNNF circuits [2]. In practice, identifying independent subformulas means checking whether some graph of the formula has several connected components. Compilers work toward splitting the graph through heuristics and strategies that guide the choices of the branching variables [18]. The idea is to favor variables that belong to certain *cutsets*, or separators, of the graph. This is explicit in a compiler like `d4` that uses (hyper)graph partitioning tools to find such cutsets. In `c2d`, the cutsets are computed beforehand from recursive graph partitioning or from a tree decomposition and organized in a data structure called a dtree that is passed on to the compiler. In a model counter like `sharpsat-TD`, cutsets are hidden in a tree decomposition used to influence the scores of the variables to branch on. The two top-down compilation algorithms we study are inspired by these compilers. The first one always branch on variables from a cutset determined by a static tree decomposition of the primal graph of the original CNF and the second dynamically looks for cutsets that are balanced separators of the graph of the current formula.

Results on the efficiency of static decomposition-based model counters compared to dynamic model counters or compilers are not novel. For instance, more than twenty years ago, for model counting, the authors of [1] showed that decomposition-based techniques like *recursive conditioning* [7] do not poly-time simulate a dynamic variant of the counting algorithm #DPLLCache. But these are results on running time, and besides they are proved for unsatisfiable formulas. This is not very useful for us since we only care about the *size* of the compiler output and, even if it takes a long time, a good compiler always returns a constant-size output on unsatisfiable inputs. In addition, we want our algorithms to be realistic in their variable selection behavior, so they cannot be completely non-deterministic. It is typically not clear for which deterministic version of #DPLLCache the results of [1] still hold. Closer to our work, the authors of [5] show lower bounds on the size of decision-DNNF circuits that are *structured-decomposable* which they argue correspond the circuits constructed by `c2d`. This is true if certain features of compilers are disabled, in particular inprocessing techniques like unit propagation, which is not something we want for our results. Yet we believe it is true that the compilers based on static decompositions construct circuits "more structured" (though not structured-decomposable) than those relying on more dynamic approaches. Our lower bounds on the static decomposition-based compilation algorithm are in fact proved using some different notion of structuredness. These lower bounds are not surprising since it is known that structured-decomposability tends to dramatically increase the size of the circuits [22]. But then it is a bit more surprising that our decomposition-based algorithm can largely outperform our dynamic algorithm, whose output is barely structured, on specific instances.

The paper is organized has follows. We start with some preliminaries in Section 2. In Section 3 we describe our framework for top-down compilation and the loose settings in which we our results fit: basically the cutset selection mechanism is strict but every other procedures (caching, inprocessing, variable selection inside the cutset) are mostly undefined, so that positive upper bound results hold even with naive procedures and negative lower bound results hold even with non-deterministic procedures. We then show in Section 4 that the static approach returns compiled forms that have some "structure". We use this in Section 5 to design functions that are easy to compile in the dynamic approach but hard for the static approach. Finally we prove that the opposite holds for other functions in Section 6.

2 Preliminaries

We use the notations $[n] = \{1, 2, \dots, n\}$ and $[0, n] = \{0, 1, \dots, n\}$.

The domain of a *Boolean variable* is $\{0, 1\}$ ($\{\mathit{false}, \mathit{true}\}$). An *assignment* to a set X of Boolean variables is a mapping $\alpha : X \to \{0, 1\}$. A *partial assignment* to X is an assignment to a subset of X. A *Boolean function* f over X maps the assignments to X to $\{0, 1\}$. α is a model of f when $\alpha \in f^{-1}(1)$. We sometimes write $f(X)$ to precise that f is a function over X. When X is not explicit we refer to it as $\mathit{var}(f)$. Given a partial assignment α to $\mathit{var}(f)$, we denote by $f|\alpha$ the Boolean function over $\mathit{var}(f) \setminus \mathit{var}(\alpha)$ whose models are the models of f consistent with α and projected onto $\mathit{var}(f) \setminus \mathit{var}(\alpha)$. Constants 0 and 1 are sometimes seen as functions over $\emptyset$.

A *literal* is a variable x or its negation $\neg x$, also written $\bar{x}$. The negated literal $\bar{\ell}$ equals x if $\ell = \bar{x}$, and equals $\bar{x}$ if $\ell = x$. A *clause* is a disjunction of literals and a CNF formula (Conjunctive Normal Form) F is a conjunction of clauses. CNF formulas are sometimes seen as sets of clauses, and we write $c \in F$ to mean $c \in \mathit{clause}(F)$. For us, $F|\alpha$ is the CNF formula obtained by removing all clauses of F containing literals satisfied by α, and all literals falsified by α from the remaining clauses. If $F|\alpha$ is empty, then it is replaced by 1. If $F|\alpha$ contains an empty clause, then it is replaced by 0. The *primal graph* G_F of F is the graph whose vertices are F's variables and such that there is an edge between x and y if and only if there is a clause $c \in F$ such that $x \in \mathit{var}(c)$ and $y \in \mathit{var}(c)$. The connected components of G_F correspond to the the largest subformulas of F that share no variable. We denote by $\mathit{components}(F)$ the set of these subformulas.

2.1 Graphs Separators and Tree Decompositions

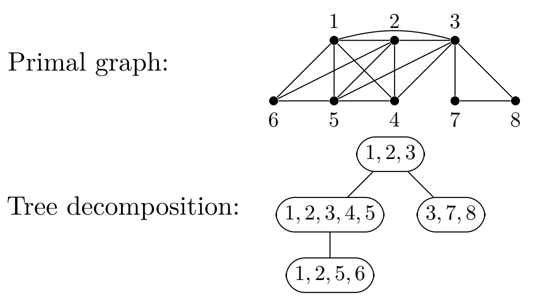
For T a tree and t and t' two nodes T, $t \leq_T t'$ means that t' is an ancestor of t or that $t' = t$, whereas $t <_T t'$ means that t' is an ancestor of t and $t \neq t'$. Let G be a graph with vertex set $V(G)$ and edge set $E(G)$. For $V \subseteq V(G)$, $G[S]$ is the graph with vertex set S and edge set $\{\{u, v\} \mid u \in S, v \in S, \{u, v\} \in E(G)\}$. We write $G - S = G[V(G) \setminus S]$. A *tree decomposition* $\mathcal{T}$ of G is a pair (T, b) where T is a rooted tree and b is a function $b : V(T) \to \mathcal{P}(V(G))$ such that

- for every $v \in V(G)$, there exists $t \in V(T)$ such that $v \in b(t)$;
- for every $\{u, v\} \in E(G)$, there exists $t \in V(T)$ such that $\{u, v\} \subseteq b(t)$;
- for every $v \in V(G)$, $T[\{t \mid v \in b(t)\}]$ is connected (so it is a tree).

The set $b(t)$ is called the *bag of* t. The width of $\mathcal{T}$ is the maximum size of a bag, so $\max_{t \in V(T)} |b(t)|$. The *treewidth* of G, noted $\mathit{tw}(G)$, is the minimum width for a tree decomposition of G minus 1. We denote by $b_\downarrow(t)$ the union of $b(t)$ and of the bags of all descendants of t, i.e., $b_\downarrow(t) = \bigcup_{t' \leq_T t} b(t')$.

Suppose G is connected. A *separator* of G, or a *cutset* of G, is a subset $S \subseteq V(G)$ such that $G - S$ has more than one connected component. For $\delta \in [0, 1]$, a δ*-balanced separator* of G is a separator S of G such that every connected component of $G - S$ has at most $\delta|V(G)|$ vertices. We denote by $s_\delta(G)$ the smallest size of a δ-balanced separator of G.

$(1\vee 2\vee 3)(\bar{1}\vee\bar{2})(\bar{2}\vee 3)(\bar{1}\vee 3)(\bar{3}\vee 7)(\bar{3}\vee 8)(7\vee 8)(1\vee 2\vee\bar{5}\vee 6)(1\vee 2\vee 5\vee 4)(\bar{1}\vee\bar{3}\vee 4\vee 5)(\bar{1}\vee\bar{3}\vee\bar{4}\vee 5)$



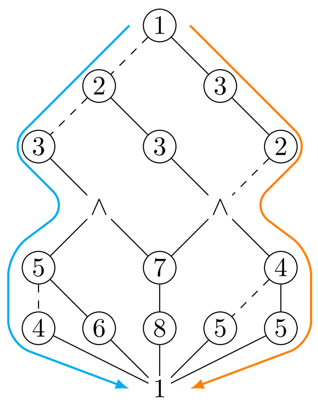


Figure 1 A CNF formula and a decision-DNNF circuit that represents it.

2.2 Decision-DNNF Circuits

A decision-DNNF (decision Decomposable Negation Normal Form) circuit is a directed acyclic graph with a single source and whose nodes are of three types: *sinks*, *decision nodes* and *decomposable* $\wedge$*-nodes*. Each node v computes a Boolean function $\langle v\rangle$ over $var(v)$.

- A sink has out-degree 0 and is labeled by a constant $c \in \{0, 1\}$. Here $\langle v\rangle = c$ and $var(v) = \emptyset$.
- A decision node v is labeled by a Boolean variable x and has two children: the 0-child v_0 and the 1-child v_1, with $x \notin var(v_0) \cup var(v_1)$. We write $v = ite(x, v_1, v_0)$ (if x then v_1 else v_0). Here $\langle v\rangle = (\bar{x} \wedge \langle v_0\rangle) \vee (x \wedge \langle v_1\rangle)$ and $var(v) = \{x\} \cup var(v_0) \cup var(v_1)$.
- A decomposable $\wedge$-node v is labeled by the conjunction symbol $\wedge$, it has children $v_1, \dots, v_k$ with $k \geq 1$ such that $var(v_i) \cap var(v_j) = \emptyset$ for every $i \neq j$. The node is interpreted as $\langle v\rangle = \langle v_1\rangle \wedge \cdots \wedge \langle v_k\rangle$ and $var(v) = var(v_1) \cup \cdots \cup var(v_k)$.

A decision-DNNF circuit C with source node v computes, or represents, the Boolean function $\langle v\rangle$ over $var(n)$. We directly see C as a Boolean function and write $var(C) = var(v)$. Graphically, for α an assignment to $var(C)$, the value $C(\alpha) = \langle v\rangle(\alpha)$ can be determined starting from v and by descending the circuit as follows: upon encountering the node u, if u is a decision node for x then continue from its $\alpha(x)$-child, if u is sink then stop, and if u is a decomposable $\wedge$-node then continue the from all its children $u_1, \dots, u_k$ in parallel. If at least one sink 0 is reached then $C(\alpha) = 0$, otherwise $C(\alpha) = 1$. Given a partial assignment β to $var(C)$, a decision-DNNF circuit for $C|\beta$ is constructed as follows: for every $x \in var(\beta)$ and every decision node $v = ite(x, v_1, v_0)$ in C, redirect all parents of v to $v_{\beta(x)}$ and delete v. Once this is done, remove all subcircuits not reachable from the source.

An example of decision-DNNF circuit is shown in Figure 1. Graphically, a decision node for x is represented with a circle labeled by x. It is connected to its 0-child by a dashed line and to its 1-child by a solid line. Only the 1-sink is represented in Figure 1. Missing outputs for decision nodes go straight to the 0-sink.

3 A Framework for Top-Down Compilation to decision-DNNF

Algorithm 1 encompasses the behavior of top-down compilers from CNF to decision-DNNF. It is largely inspired from [17]. The priority of the algorithm is to split the primal graph G_F of the input formula F. This is done by selecting a cutset of the graph, that is, a set of vertices/variables whose removal leaves the graph disconnected. Assigning a variable to 0 or 1 removes at least this variable from the graph. The cutset variables are assigned in all possible ways until the graph is disconnected, which may happen before all are assigned. Every variable assignment adds a decision node to the circuit (line 11). Between two successive variable assignments, the algorithm checks whether the graph is disconnected and, if so, a decomposable $\land$-node is created and the subformulas for each components are dealt with independently (line 7). The cutset is reset to $\emptyset$ for these subformulas to notify that a new cutset must be computed. Due to its recursive nature, `Compile` requires two arguments: F and a subset S of F's variables that corresponds to what remains of the cutset. When S is empty (which is the case initially) we write `Compile`(F) instead of `Compile`$(F, \emptyset)$.

Cutset selection corresponds to the procedure `selectCutset`. It is the one procedure where the two algorithms studied in this paper behave differently. In the *dynamic approach*, `selectCutset`(F) returns a balanced separator of G_F whose size is minimal, or close to minimal. In the *static approach*, we have access to a tree decomposition $\mathcal{T}$ of G_F or of a supergraph of G_F and `selectCutset`(F) returns the bag of a highest node of $\mathcal{T}$ which has a non-empty intersection with $var(F)$. There will always be a unique such node in our settings. We give names to the two variants of `Compile`.

- `Compile`$_{d,\epsilon}(F)$ is the algorithm `Compile` where `selectCutset`(F) returns a 2/3-balanced separator of G_F of size at most $(1+\epsilon)s_{2/3}(G_F)$.
- `Compile`$_s(F, \mathcal{T})$ is the algorithm `Compile` with an extra argument: a tree decomposition $\mathcal{T}$ that remains constant through the whole algorithm, and where `selectCutset`(F) returns a highest bag of $\mathcal{T}$ that has a non-empty intersection with $var(F)$.

The dynamic approach requires solving a hard problem (graph balanced-partitioning) several time during compilation while the static approach requires a hard preprocessing step, namely computing a tree decomposition whose width is close to minimal. We disregard running times in this paper, so we just ignore the complexity of these problems. However, to be a bit closer to reality we try not to put too much constrain on `selectCutset`, hence the requirement that the size of the cutset in the dynamic approach is minimal *up to a fixed constant factor*. One forces the dynamic compile algorithm to find a minimal-size cutset by setting $\epsilon = 0$. We will write `Compile`$_d$ instead of `Compile`$_{d,0}$. Similarly, for the static approach, we will allow tree decompositions of width minimal *up to a fixed constant factor*. We leave tie-breaking non-deterministic when several candidates exist for the output of `selectCutset`. We stress out that the two variants of `Compile` work on the *primal* graph. Practical compilers may choose other graphs, like `d4` which uses a *dual hypergraph* of F [17].

Variable selection corresponds to the procedure `selectVariable` for selecting the next variable to assign in the cutset. We leave this procedure undefined. One can select variables in a predefined order, or use frequency-based heuristic, or heuristics influenced by the outcome of the algorithm on previous branches like VSIDS, or a non-deterministic oracle, etc. Our results are agnostic to this procedure. The only rule is to *not select a variable outside of the cutset*. For the static approach, this is actually a deviation from practical tree-decomposition-based compilers and model counters. For instance, `sharpSAT-TD` [16] selects variables based on a score computed using the depth of the variables in the tree decomposition but also their frequency and VSIDS scores. On the one hand, the depth component of the score makes it more likely to select a variable appearing in the highest bag of the decomposition, which corresponds to our cutset in the static approach. On the other hand, the VSIDS and frequency components can force the selection of a variable outside of the cutset. It has been noticed that the depth component is dominant in practice and that the other components mostly serve as a tie-breaking mechanism [16, Section 5]. So we think our model is quite realistic. Though we do not know if it is for the functions that we use to prove our results.

Caching allows to avoid constructing a decision-DNNF for the same formula twice. By default, `cache`(F) equals `nil` for every formula F. We consider two caching variants: one realistic, the other idealized. The realistic caching is *syntactical*: in Line 5, `cache`$(F) \neq$ `nil` means that the formula F has already been seen in previous calls to the algorithm. The idealized caching is *semantical*: in Line 5, `cache`$(F) \neq$ `nil` means that some formula F' logically equivalent to F, with $var(F') \subseteq var(F)$, has been seen in previous calls (for instance $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_3)$ and $F' = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3)$), and then `cache`$(F)$ is defined as `cache`(F'). The condition $var(F') \subseteq var(F)$ is here to avoid situations where decomposability of $\wedge$-nodes would be compromised because of caching. We write `Compile`$[smc](\dots)$ to precise that $\underline{s}$e$\underline{m}$antical $\underline{c}$aching is used. Otherwise, syntactical caching is used (no caching is not allowed).

Inprocessing simplifies the formula between every two variable selections. This is the procedure `process`(F) at Line 2. Here it returns a modified formula for F and a term (a conjunction of literals) τ. A classical procedure is *unit propagation*: while there is a *unit clause* $\ell \in F$, ℓ is added to the term τ and F is replaced by $F|\ell$. Unit propagation is linear-time, but we can also have more complex processing procedures like *satisfiability checking*: if F is unsatisfiable, then it is replaced by 0. The stronger procedure we consider is *backbone identification*: τ contains all literals ℓ such that $F \models \ell$ and F is replaced by $F|\tau$. Note that backbone identification subsumes both unit propagation and satisfiability checking. Indeed if F is unsatisfiable, then $F \models x$ and $F \models \bar{x}$ for every x and we just assume `process`(F) returns $(0, \{x, \bar{x} | x \in var(F)\})$. A compiler may implement several processing techniques at the same time. In practice NP-hard processing procedures require calling an external SAT solver. Again, we just disregard the running time of `process` and assume it is sound and complete. We write `Compile`$[bb](\dots)$ when $\underline{b}$ack$\underline{b}$one identification is enabled (and a fortiori unit propagation and satisfiability checking). Without $[bb]$ inprocessing is disabled (even unit propagation), then `process`(F) returns $(F, \emptyset)$. When τ is not empty, then it is added to the output with a $\wedge$-node at line 7 or 11. For instance, if $F = (x_1) \wedge (\bar{x}_1 \wedge \bar{x}_2) \wedge (x_2 \vee \bar{x}_1 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5 \vee x_6)$, and backbone identification is enabled, then `process`$(F) = ((x_4 \vee x_5 \vee x_6) \wedge (\bar{x}_4 \vee \bar{x}_5 \vee x_6), x_1 \wedge x_2 \wedge x_3)$ and the algorithm goes through lines 9, 10 and 11. Say x_4 is selected line 10, then line 11 returns the following:

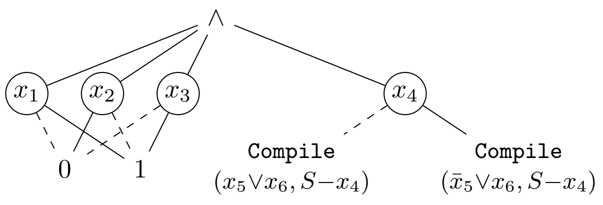

$\wedge$
x_1
x_2
x_3
x_4
0
1
Compile $(x_5 \vee x_6, S - x_4)$
Compile $(\bar{x}_5 \vee x_6, S - x_4)$

Figure 2

where each literal of $\tau = x_1 \wedge x_2 \wedge x_3$ is converted into a decision node to make sure that the resulting circuit is a decision-DNNF. Admittedly, in this paper we are far from exhaustive when it comes to inprocessing, for instance we do not consider literal equivalences detection [17], vivification [21], elimination of redundant clauses, etc.

4 Decision-DNNF Organized by Tree Decompositions

The output of `Compile`$_s(F, \mathcal{T})$ is a decision-DNNF circuit *organized by* $\mathcal{T}$. We introduce this notion in this section. In a decision-DNNF circuit, a *path* from the source to a sink is a sequence of nodes $(v_1, v_2, \dots, v_k)$ with v_1 the source, v_{i+1} a child of v_i, and v_k a sink.

Algorithm 1 The general `Compile` procedure.

```
1:  input: a CNF formula $F$, a set of variable $S \subseteq var(F)$
2:  $(F, \tau) \leftarrow$ process$(F)$                          // $\tau$ is a conjunction of literals
3:  $S \leftarrow S \setminus var(\tau)$
4:  if $F = 0$ or $F = 1$ then return $F$
5:  if cache$(F) \neq$ nil then return cache$(F)$                          // cache check
6:  if $F$ has more than one connected component then
7:      $C \leftarrow \tau \wedge \bigwedge_{H \in components(F)}$ Compile$(H, \emptyset)$        // create a decomposable $\wedge$-node
8:  else
9:      if $S = \emptyset$ then $S \leftarrow$ selectCutset$(F)$
10:     $x \leftarrow$ selectVariable$(S, F)$
11:     $C \leftarrow \tau \wedge ite(x,$ Compile$(F|x, S - x),$ Compile$(F|\bar{x}, S - x))$      // create a decision node
12: end if
13: cache$(F) \leftarrow C$                                               // cache update
14: return $C$
```

Note that, when $i < k$, v_i may be a decision node or an $\wedge$-node. We use $\cdot$ to denote path concatenation. For instance $(v_1, v_2, v_3, v_4) = (v_1, v_2) \cdot (v_3, v_4)$. A variable x appear in a path p if p contains a decision node for x. For $\mathcal{T} = (T, b)$ and $x \in var(F)$ we denote by t_x the highest node of T whose bag contains x, that is, $x \in b(t_x)$, and there is no $t \in V(T)$ such that both $t_x <_T t$ and $x \in b(t)$.

▶ **Definition 1.** *Let F be a CNF formula and let $\mathcal{T} = (T, b)$ be a tree decomposition of G_F. A path p in a decision-DNNF circuit C is* organized *by $\mathcal{T}$ when, for every two decision nodes v_x and v_y on p for the variables x and y, respectively, v_x appears before v_y in p only if $t_y \leq_T t_x$. C is organized by $\mathcal{T}$ when all its paths are organized by $\mathcal{T}$.*

For example, the decision-DNNF circuit of Figure 1 is organized by the tree decomposition shown in the same figure. Readers well-versed in knowledge compilation may know the concept of *structured-DNNF circuits* [22] and may see similarities with our circuits organized by tree decompositions. But they should also note that the circuit shown in Figure 1 is not *structured-decomposable*. Indeed, a structured-DNNF circuit cannot have the two paths highlighted in the figure since they both contain the variables 2, 3, 4, 5 but in different order.

We are show that the output of $\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ is organized by $\mathcal{T}$. We call *residual component of F* any $R \in components(F|\alpha)$ where α is a partial assignment to $var(F)$. Note that G_R is connected.

▶ **Lemma 2.** *Let F be a CNF formula, let $\mathcal{T} = (T, b)$ be a tree decomposition of G_F and let R be a residual component of F such that $var(R) \neq \emptyset$. Then there is a unique highest $t_R \in T$ such that $var(R) \cap b(t_R) \neq \emptyset$.*

Proof. Let $t_1, \dots, t_k$ be the highest nodes in T whose bags intersect $var(R)$. Suppose, toward a contradiction, that $k > 1$. Then let t be the least common ancestor of $t_1, \dots, t_k$. t has at least two children c_ℓ and c_r such that some t_i is a descendant of c_ℓ (or is c_ℓ itself) and some t_j, $j \neq i$, is a descendant of c_r (or is c_r itself). By assumption, $var(R) \cap b(t) = \emptyset$. As a general property of tree decompositions, $b(t)$ is a vertex separator of $G_F[b_\downarrow(t)]$ such that no component of $G_F[b_\downarrow(t)] - b(t)$ contains at the same time variables from $b_\downarrow(c_\ell) \setminus b(t)$ and variables from $b_\downarrow(c_r) \setminus b(t)$. So, since G_R is a subgraph of $G_F[b_\downarrow(t)] - b(t)$, we cannot have that G_R is connected and intersect both $b(t_i)$ and $b(t_j)$. This is a contradiction, so $k = 1$. ◀

▶ **Lemma 3.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. $\texttt{Compile}_s(F, \mathcal{T})$ returns a decision-DNNF circuit organized by $\mathcal{T}$ and representing F.*

Proof. Here we have no inprocessing and only syntactical caching. Let $\mathcal{T} = (T, b)$ and let C be the output of $\texttt{Compile}_s(F, \mathcal{T})$. The statement follows from Lemma 2 and the fact that, given any residual component R of F, $\texttt{Compile}_s(R, \mathcal{T})$ constructs a decision-DNNF circuit over $var(R)$. Indeed, suppose C contains a path that violates Definition 1, that is, the path contains a decision node v_x for a variable x before the decision node v_y for the decision node y whereas $t_y \not\leq_T t_x$. Let R be the residual component of F for which $\texttt{Compile}_s(R, \mathcal{T})$ constructed the node v_x. We have $\{x, y\} \subseteq var(R)$. By Lemma 2, $\texttt{Compile}_s(R, \mathcal{T})$ must select a variable from $b(t_R)$ and, by uniqueness of t_R, t_R is either t_y or a strict ancestor of both t_x and t_y. In both cases $x \notin b(t_R)$ so $\texttt{Compile}_s(R, \mathcal{T})$ cannot select x and thus does not construct v_x. A contradiction. ◀

▶ Remark 4. Decision-DNNF circuits organized by tree decompositions lie between general decision-DNNF circuits and decision-structured-DNNF circuits (see for instance [5]). In the extreme case where $\mathcal{T}$ is a single bag containing all variables, every decision-DNNF circuit is organized by $\mathcal{T}$. Thus, the "one-bag" tree decomposition, while useless in practice, is the best for $\texttt{Compile}_s$ (more possibility for the output). But this is only because the only constraint for `variableSelect` is to select a variable from the bag. Using the "one-bag" tree decomposition essentially means lifting that constraint.

Lemma 3 still holds when non-deterministic caching is enabled.

▶ **Lemma 5.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. Algorithm $\texttt{Compile}_s[smc](F, \mathcal{T})$ returns a decision-DNNF circuit organized by $\mathcal{T}$ that represents F.*

Proof sketch. We start with a trivial but key observation. Let $p = (v_1, \dots, v_k)$ and $p' = (v'_1, \dots, v'_h)$ be two paths organized by $\mathcal{T}$. For some $i \in [k-1]$ and some $j \in [h-1]$ let $p_1 = (v_1, \dots, v_i)$, $p_2 = (v_{i+1}, \dots, v_k)$, $p'_1 = (v'_1, \dots, v'_j)$, $p'_2 = (v'_{j+1}, \dots, v'_h)$. If for every variable x appearing in p_1 and every variable y appearing in p'_2 we have $t_y \leq_T t_x$, then the path $p_1 \cdot p'_2$ is organized by $\mathcal{T}$.

Now consider a run of $\texttt{Compile}_s[smc](F, \mathcal{T})$. Suppose every paths constructed up to the call $\texttt{Compile}_s[smc](R, S, \mathcal{T})$, with R a residual component of F, are organized by $\mathcal{T}$. Let p be the path corresponding to the branch that lead to $\texttt{Compile}_s[smc](R, S, \mathcal{T})$ and suppose that for all $y \in var(R)$ and all $x \in var(p)$ we have $t_y \leq_T t_x$. Assume $\texttt{Compile}_s[smc](R, S, \mathcal{T})$ identifies that there exists a CNF formula R' with $C' = \texttt{cache}(R') \neq \texttt{nil}$ such that R is logically equivalent to R' and such that $var(R') \subseteq var(R)$. Since $var(C') \subseteq var(R')$, for all $y \in var(C')$ and $x \in var(p)$, $t_y \leq_T t_x$ holds true. So, by the previous observation, since all paths in C' are organized by $\mathcal{T}$, concatenating a path of C' to p does not create any path not organized by $\mathcal{T}$. So, if $\texttt{Compile}_s(F, \mathcal{T})$ returns a circuit organized by $\mathcal{T}$, then so does $\texttt{Compile}_s[smc](F, \mathcal{T})$. ◀

Now let us further assume that `process` does backbone identification. Given F, $\texttt{process}(F)$ returns $\tau = ite(\ell_1, 1, 0) \wedge \dots \wedge ite(\ell_k, 1, 0)$ where $F \models \ell_i$ for every literal ℓ_i ($k = 0$ and $\tau = 1$ if no such literal exists), and replaces F by $F|\ell_1 \dots \ell_k$. When $k > 0$ the algorithm calls the decision nodes of τ are conjoined to the circuit. We say that these decision nodes have been *inferred from backbone identification*. Let $p = (v_1, v_2, \dots, v_k)$ be a source-to-sink path in C. There can be only one decision node inferred from backbone identification in this path: v_{k-1} (v_k is a sink). Every decision node v_i appearing before v_{k-1} on this path is for a variable y_i that has been selected from a bag of the tree decomposition. Let us call x the variable for v_{k-1}. For any given y_i we have $t_x \leq_T t_{y_i}$. Indeed, let R be the connected residual formula of F such that the call $\texttt{Compile}_s(R, \mathcal{T})$ constructed v_i. By Lemma 2, since $x \in var(R)$, we have that $t_x \leq_T t_R$, and since y_i was selected we also have

$t_R = t_{y_i}$. So we see that adding backbone identification, and a fortiori unit propagation or satisfiability testing, does not change the fact that $\texttt{Compile}_s$ returns a decision-DNNF organized by $\mathcal{T}$.

▶ **Lemma 6.** *Let F be a CNF formula and $\mathcal{T}$ be a tree decomposition of G_F. Algorithm* $\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ *returns a decision-DNNF circuit organized by $\mathcal{T}$ that represents F.*

We finish this section with a result on the manipulation of decision-DNNF circuits organized by tree decomposition which will be needed in proofs to come. For $S \subseteq V(G)$ we denote by $\mathcal{T} - S$ the tree decomposition of $G - S$ obtained from $\mathcal{T}$ by removing every vertex of S from its bags. Formally, when $\mathcal{T} = (T, b)$, we have $\mathcal{T} - S = (T, b')$ with $b'(t) = b(t) \setminus S$.

▶ **Lemma 7.** *conditioning Let F be a CNF formula let C be a decision-DNNF circuit organized by the tree decomposition $\mathcal{T}$ of G_F. Let α be a partial assignment to F. Then $C|\alpha$ is a decision-DNNF circuit organized by the tree decomposition $\mathcal{T} - var(\alpha)$ of $G_F - var(\alpha)$.*

Proof. We prove the statement when α is an assignment to a single variable x. The lemma then follows by induction. Then $C|\alpha$ is obtained by replacing every decision node $ite(x, v_1, v_0)$ by $v_{\alpha(x)}$. It is readily verified that the paths in $C|\alpha$ are subpaths of paths of C: let $v = ite(x, v_1, v_0)$, then a path $p_0 \cdot (v, v_{\neg\alpha(x)}) \cdot p_1$ in C is not kept in $C|\alpha$ while a path $q = q_0 \cdot (v, v_{\alpha(x)}) \cdot q_1$ in C becomes $q' = q_0 \cdot (v_{\alpha(x)}) \cdot q_1$ in $C|\alpha$. For every variable different from x, the highest node whose bag contains it in $\mathcal{T} - \{x\}$ is the same as in $\mathcal{T}$, so if q is organized by $\mathcal{T}$ then q' is organized by $\mathcal{T} - \{x\}$. ◀

5 Hard Functions for Static Top-Down Compilation

In this section we show that there are CNF formulas that are hard for $\texttt{Compile}_s$ but easy for $\texttt{Compile}_d$.

▶ **Theorem 8.** *There is an infinite class $\mathcal{F}$ of CNF formulas and a constant $\delta \in (0, 1]$ such that, for every $F \in \mathcal{F}$ over n variables and every tree decomposition $\mathcal{T}$ of G_F of width $O(tw(G_F))$, the following holds:*
- $\texttt{Compile}_d(F)$ *returns a decision-DNNF circuit of size $n^{O(1)}$;*
- $\texttt{Compile}_s[smc, bb](F, \mathcal{T})$ *returns a decision-DNNF circuit of size $2^{\Omega(n^\delta)}$.*

One can see that, as n increases, $tw(G_F)$ has to become negligible compared n, because if $tw(G_F) = \Omega(n)$ then the "one-bag" tree decomposition becomes an option and $\texttt{Compile}_s$ essentially finds the smallest decision-DNNF (see Remark 4). One can also guess that $tw(G_F)$ cannot bounded by a constant for all $F \in \mathcal{F}$, because $\texttt{Compile}_s$ should be able to create circuits on size $2^{O(tw(G_F))}n^{O(1)}$. Grid graphs are convenient to get a treewidth that is large enough and yet vanishingly small compared to n. We start with some preliminaries on grid graphs. Then we describe our functions and explain why $\texttt{Compile}_d$ is effective on them. Finally, we prove the lower bound for $\texttt{Compile}_s$.

5.1 Grid Graphs and Spine Graphs

The grid graph $grid_{n,m}$ contains $n \times m$ vertices $\{x_{ij} \mid i \in [n], j \in [m]\}$ connected in n rows and m columns. Grid graphs have a nice well-known properties that we are going to use several time.

▶ **Lemma 9.** *For every fixed $\delta \in [0, 1)$, the $grid_{n,n}$ as no δ-balanced separator of size $o(\sqrt{n})$.*

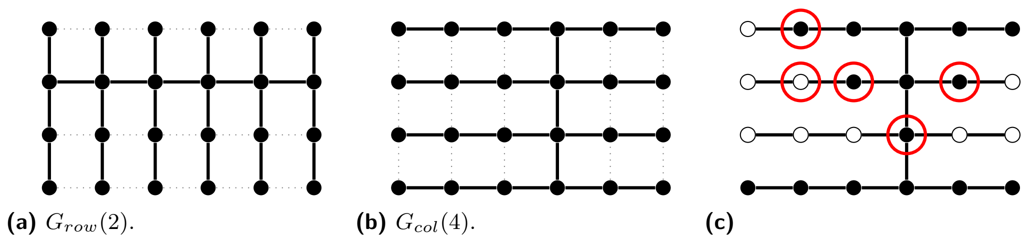

(a) $G_{row}(2)$. (b) $G_{col}(4)$. (c)

Figure 3 Spine subgraphs of $grid_{4,6}$.

Proof. For $S \subseteq V(grid_{n,n})$, Let $\partial(S) = \{\{u,v\} \in E(grid_{n,n}) \mid u \in S, v \notin S\}$. By [3, Lemma 3], for every S of size $|S| \leq n^2/2$, $|\partial(S)| \geq \min(n, 2\sqrt{|S|})$ holds true. Suppose S is a δ-balanced separator of $grid_{n,n}$. Let $V_1, V_2, \dots$ be the components of $grid_{n,n} - S$. There is V', a union of some components of $grid_{n,n} - S$ such that $|V'| \leq n^2/2$ and $|V'| = \Omega(n^2 - |S|)$. Indeed suppose there is a component V_j of size $\delta n^2/2 \leq |V_j| \leq \delta n^2$. If $|V_j| \leq n^2/2$ we can choose $V' = V_j$ and we are done. If $|V_j| \geq n^2/2$ then we choose $V' = \bigcup_{i \neq j} V_i$ and obtain $|V'| \geq n^2 - |S| - |V_j| \geq (1-\delta)n^2 - |S|$. But when V_j does not exist, we let V' be $V_1 \cup \dots \cup V_i$ for the largest i such that $|V_1 \cup \dots V_i| \leq (n^2 - |S|)/2$, then we have that $|V'| + |V_{i+1}| \geq (n^2 - |S|)/2$ and $|V'| \geq ((1-\delta)n^2 - |S|)/2$. Now, $|V'| \leq n^2/2$ implies that $|\partial(V')| \geq \min(n, 2\sqrt{|V'|}) = \Omega(\sqrt{n^2 - |S|})$. It follows that if $|S| = o(\sqrt{n})$, then $|\partial(V')| \geq \Omega(n)$. So removing $\Omega(n)$ edges is needed to disconnect V' from the rest of the grid. But every vertex in $grid_{n,n}$ has degree at most 4, so removing the vertex set S cannot remove more than $4|S| = o(\sqrt{n})$ edges. So removing S is not enough to disconnect V' from the rest of the grid, a contradiction. ◀

For n and m fixed, we write $row_i = \{x_{i1}, \dots, x_{im}\}$ and $col_j = \{x_{1j}, \dots, x_{nj}\}$. An edge $\{x_{ij}, x_{i\,j+1}\}$ is called an edge of the ith row. An edge $\{x_{ij}, x_{i+1\,j}\}$ is called an edge of the jth column.

▶ **Definition 10** (Spine subgraphs). *A* spine subgraph *of $G = grid_{n,m}$ is a subgraph $G_{row}(i)$, $i \in [n]$ whose edge set comprises all column edges plus all edges of the ith row, or a subgraph $G_{col}(j)$, $j \in [m]$, whose edge set comprises all row edges plus all edges of the jth column. The unique row (resp. column) of $G_{row}(i)$ (resp. $G_{col}(j)$) is called its* spine.

Examples of spine subgraphs are shown in Figures (3a) and (3b). For an unspecified spine subgraph H, we refer to its spine as *spine*(H). For every $i, k \in [n]$ and $j, \ell \in [m]$, the vertex x_{ij} is at distance $|i-k|$ from $G_{row}(k)$'s spine and at distance $|j-\ell|$ from $G_{col}(\ell)$'s spine.

▶ **Definition 11.** *For $S \subseteq V(grid_{n,m})$ and a spine subgraph H of $grid_{n,m}$, the vertex x_{ij}* sees H's spine despite S *if there is no vertex of $S \setminus \{x_{ij}\}$ on the shortest path connecting x_{ij} to H's spine.*

An example is shown in Figure (3c). The vertices of S are circled in red. The vertices that see the spine despite S are in black. The vertices that do not are in white. Note that not all vertices in S see the spine.

▶ **Lemma 12.** *Let $c > 1$ be a constant, $G = grid_{n,m}$ with $m \geq (3c+1)n$, and $S \subseteq V(G)$ with $n/3 \leq |S| \leq cn$. For n large enough, there is a spine subgraph H of G such that no vertex of S is at distance fewer than 2 from spine(H) and such that at least $\Omega(\sqrt{n})$ vertices of S see spine(H) despite S.*

Proof. Let $I = \{i \mid \exists j, x_{ij} \in S\}$ and let $J = \{j \mid \exists i, x_{ij} \in S\}$ be the indexes of the rows and columns, respectively, that intersect S.

If $|I| < \sqrt{n}/3$, then at least one row indexed in I contains $\sqrt{n}$ vertices of S. So $|J| \geq \sqrt{n}$. Moreover, $|I| < \sqrt{n}/3$ implies that at least $n - \sqrt{n}/3$ rows do not intersect S and thus at least $\frac{1}{3}(n - \sqrt{n})$ rows do not intersect S and are such that they neighboring rows do not intersect S (the neighboring rows of row_i being row_{i+1} and row_{i-1}). Let H be the spine subgraph $G_{row}(i)$ for any of these rows. Then all vertices of S are at distance at least 2 from $spine(H)$. Finally, for every $j \in J$, there is one vertex of $S \cap col_j$ that sees $spine(H)$ despite S.

Now suppose $|I| \geq \sqrt{n}/3$. Since $|S| \leq cn$, at least one of the m columns does not intersect S and is such that its neighboring columns do not intersect S either. Let H be the spine subgraph for that column. Then $spine(H) \cap S = \emptyset$ and all vertices of S are at distance at least 2 from $spine(H)$. Finally for every $i \in I$, there is one vertex of $S \cap row_i$ that sees $spine(H)$ despite S. ◀

5.2 The Hard Functions

We assume the tree decompositions given to $\texttt{Compile}_s$ have minimal width up to a constant factor $\rho \geq 1$ and we write $n' = \lceil 12\rho + 1 \rceil n$. There are $O(n)$ spine subgraphs of $grid_{n,n'}$. In this section, we call them $H_1, H_2, \dots$ For each H_i we have a CNF formula

$$F_{H_i} = \bigwedge_{\{x,y\} \in E(H_i)} (x \vee y).$$

We introduce $s = \log(n) + O(1)$ variables $Z = \{z_0, \dots, z_{s-1}\}$ such that 2^s is greater than the number of spine subgraphs. The Z-variables are used as selectors. Every assignment to Z is interpreted as a number between 1 and 2^s by $w(Z) = 1 + \sum_{k=0}^{s-1} z_k 2^k$. We denote by $(w(Z) \neq i)$ the clause satisfied exactly by the assignments α to Z verifying $w(\alpha) \neq i$. Let X be the variables/vertices of $grid_{n,n'}$. We define

$$SelectSpine_n(X, Z) = \bigwedge_{i \in [2^s]} \bigwedge_{c \in F_{H_i}} ((w(Z) \neq i) \vee c).$$

We then introduce a copy of $grid_{n,n'}$ over a new set of variables/vertices X', and we define

$$F_n(X, X', Z) = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z). \tag{1}$$

The F_n are our hard functions. Notice that the same selectors are used for the two $SelectSpine$ formulas. The reason for copying the variables is to help $\texttt{Compile}_d$ chooses Z as its first cutset. Indeed, the primal graph of F_n looks like Figure 4. Intuitively, Z is the smallest 2/3-balanced separator of this graph: only $O(\log(n))$ vertices, whereas cutting through one of the two grids requires $\Omega(n)$ vertices. So, $\texttt{Compile}_d$ starts by assigning all variables of Z in every possible way. This represents only $O(n)$ branches that each leads to a formula $F_H(X) \wedge F_{H'}(X')$ for two spine subgraphs H and H' over disjoint set of variables. The algorithm create a decomposable $\wedge$-node to deal with F_H and $F_{H'}$ independently and easily compile them into small decision-DNNF circuits.

▶ **Lemma 13.** *For every $n > 0$, $\texttt{Compile}_d(F_n)$ returns a decision-DNNF circuit of size $n^{O(1)}$, where $F_n(X, X', Z) = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z)$ is defined as in (1).*

Proof sketch. The primal graph of $F_n = SelectSpine_n(X, Z) \wedge SelectSpine_n(X', Z)$ is connected. So $\texttt{Compile}_d(F_n)$ first computes a minimal-size 2/3-balanced separator of G_{F_n}. Z is a 2/3-balanced cutset of G_{F_n} so a minimal-size 2/3-balanced cutset S of G_{F_n} contains no more than $O(\log(n))$ variables.

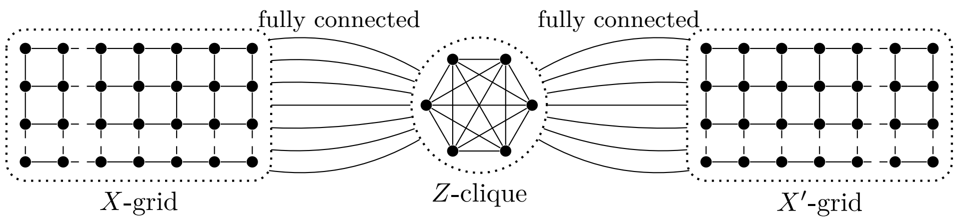


Figure 4 The primal graph of $\mathit{SelectSpine}_n(X, Z) \wedge \mathit{SelectSpine}_n(X', Z)$.

Suppose S does not contain Z in its entirety. By Lemma 9, there are no 3/4-balanced separator of size $O(\log(n))$ of an $n \times n$ grid. Let G_X be the X-grid and $G_{X'}$ be the X'-grid. Both $G_X - S$ and $G_{X'} - S$ contain a connected component of size at least $3nn'/4$. Let Γ_X and $\Gamma_{X'}$ be these components. But then Γ_X and $\Gamma_{X'}$ are connected to a vertex $z \in Z \setminus S$ in $G_{F_n} - S$ and thus $G_{F_n} - S$ has a connected component of size at least $6nn'/4$. The number of variables of F_n is $2nn' + O(\log(n))$ and $6nn'/4 > 4nn'/3 + O(\log(n))$ for n large enough. So S is not a 2/3 balanced separator if it does not contain Z. And since Z is a 2/3-balanced cutset of G_{F_n}, we have $S = Z$.

As long as one Z-variable is left unassigned, the primal graph of the formula remains connected and, since there is no inprocessing mechanisms here, $\texttt{Compile}_d(F_n)$ keeps assigning all Z-variables in all possible way without creating decomposable $\wedge$-nodes. This amounts for $O(n)$ branches in the algorithms. When, at the end of a branch, all Z-variables are assigned the algorithm makes a call $\texttt{Compile}_d(F_H \wedge F_{H'})$ with H a spine subgraph of the X-grid and H' a spine subgraph of the X'-grid. A decomposable $\wedge$-node is created and the algorithm calls $\texttt{Compile}_d(F_H)$ and $\texttt{Compile}_d(F_{H'})$. H and its connected subgraphs all have 2/3-balanced separators of constant size, so $\texttt{Compile}_d(F_H)$ only need $O(\log(|H|)) = O(\log(n))$ recursive calls to finish. ◀

5.3 Lower Bounds for $\texttt{Compile}_s$

We now prove a $2^{\Omega(\sqrt{n})}$ lower bound on the size of the decision-DNNF circuits returned by $\texttt{Compile}_s$ for F_n when given a tree decomposition $\mathcal{T} = (T, b)$ of G_{F_n} of width at most $\rho \cdot tw(G_{F_n})$.

We give some intuition for why the lower bound holds true. Just like $\texttt{Compile}_d$, $\texttt{Compile}_s$ can assign the Z-variables first if they are in the highest bags of $\mathcal{T}$, then it would find $O(n)$ subformulas of the form $F_H \wedge F_{H'}$ with H a spine subgraph of the X-grid and H' a spine subgraph of the X'-grid. In such a situation, $\texttt{Compile}_s$ will create $\wedge$-nodes and compile F_H and $F_{H'}$ separately. Let $H_1, H_2, \dots, H_{2n}$ be the spine subgraphs of the X-grid. The problem is that $\texttt{Compile}_s$ uses the *same* tree decomposition to compile $F_{H_1}, F_{H_2}, \dots, F_{H_{2n}}$. For each $i \in [2n]$, there is indeed a tree decomposition $\mathcal{T}$ of the X-grid such that $\texttt{Compile}_s(F_{H_i}, \mathcal{T})$ constructs a small circuit, but there is no tree decomposition of the X-grid that simultaneously give a small circuit for all i. And $\texttt{Compile}_s$ is stuck with a unique tree decomposition of G_{F_n} (which contains a tree decomposition of the X-grid), so for some F_{H_i} the circuit constructed will be large.

Of course the lower bound has to be proved even in cases where the Z variables are not assigned first. For $\mathcal{T}$ fixed, we give a spine subgraph H such that F_H does not admit small decision-DNNF circuits organized by $\mathcal{T}$. Then we will just use Lemma 7 to extract in from a decision-DNNF circuit C representing F_n and organized by $\mathcal{T}$, a decision-DNNF circuit C' representing F_H and organized by $\mathcal{T}$. If C' is large, then C must be large and we will be done.

For a given path p of T we let $b(p) = \bigcup_{t \in p} b(t)$.

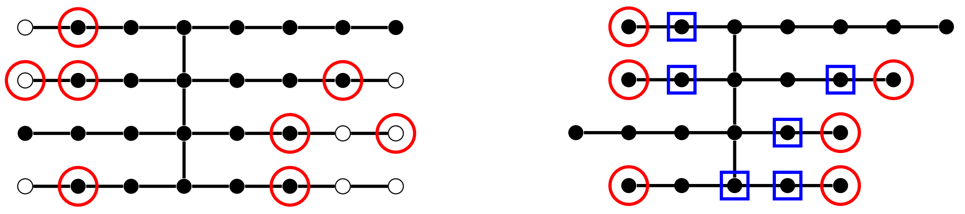
▶ **Lemma 14.** *Let $\mathcal{T}$ be a tree decomposition of width $w > 0$, there is a path p from its root to a node such that $w \leq |b(p)| \leq 2w$.*

Proof. Let t be one of the highest nodes of T with $|b(t)| = w$ and let q be the path from T's root to t. For every node $s \in q$, let q_s be the path from T's root to s. For two consecutive s, s' in q, s before s', we have $|b(q_{s'})| - |b(q_s)| \leq w$. So, since $|b(q_t)| = |b(q)| \geq |b(t)| = w$, there must be a node $s \in q$ such that $w \leq |b(q_s)| \leq 2w$. ◀

It is known that $tw(grid_{n,n}) = n$ and that there are tree decompositions of $grid_{n,n'}$ of width $n+1$, so $tw(grid_{n,n'}) = n$. It is not hard to see that $tw(F_n)$ is at least n and at most $n + O(\log(n))$ and thus, the width of $\mathcal{T}$ is between n and $2\rho n$ for n large enough. The set $b(p)$ from Lemma 14 for $\mathcal{T}$ then contains between n and $4\rho n$ variables from X, X' and Z. There are only $O(\log(n))$ variables in Z so $|b(p) \cap X| \geq n/3$ or $|b(p) \cap X'| \geq n/3$ (for n large enough). Assuming $|b(p) \cap X| \geq n/3$ holds and setting $S = b(p) \cap X$, we claim that the spine subgraph H, given by Lemma 12 for S in the X-grid, is such that F_H has no small decision-DNNF circuit organized by $\mathcal{T}$.

▶ **Lemma 15.** *For every tree decomposition $\mathcal{T}$ of width at most $2\rho n$ of G_{F_n}, there is a spine subgraph H of the X-grid or of the X'-grid such that all decision-DNNF circuits that represent F_H and are organized by $\mathcal{T}$ have size $2^{\Omega(\sqrt{n})}$.*

The proof appears in the next section. We try to give a high-level idea, suppose H is the spine subgraph represented on the left of Figure 5a. The S-vertices/variables are circled in red. Since H is found using Lemma 12, many S-vertices see $spine(H)$ despite S and all S-vertices are at distance at least 2 from $spine(H)$. If we have a decision-DNNF circuit C organized by $\mathcal{T}$ for F_H, then we can assign all variables that do not see the spine to 1 (the white vertices on the figure) and we obtain another decision-DNNF circuit C' organized by $\mathcal{T}$ (by Lemma 7) for the formula $F_{H'}$ where H' is shown on the right. Only the S-variables that see the spine remain, call them S'. In C', these variables are assigned first. But one cannot disconnect H' by removing any subset of S'. Also, unit propagation or backbone identification can only infer the value for the direct neighbors of the S' variables, but these neighbors are not on the spine and so their removal along with S' will let the graph connected. So we are essentially condemn to assign all the S'-variables and sometimes their neighbors in H', and this create $2^{\Omega(|S'|)} = 2^{\Omega(\sqrt{n})}$ nodes in C'.


(a) S-vertices are circled. Black vertices see the spine despite S.

(b) The induced subgraph $H[V]$. S'-vertices are circled. $\eta(S')$-vertices are squared.

Figure 5

So to prove Theorem 8, the upper bound on $\texttt{Compile}_d$'s output is Lemma 13, and the lower bound on $\texttt{Compile}_s$'s output follows from Lemmas 6, 7 and 15 since, from the output of $\texttt{Compile}_s[smc, bb](F_n, \mathcal{T})$, one obtains a decision-DNNF organized by $\mathcal{T}$ for F_H by assigning the Z-variables to select only the clauses of F_H.

5.4 Proof of Lemma 15

Before starting, let us make a simple observation on the formulas defined over graphs like F_H. We say that a formula F can be *decomposed* if it can be written as $f \wedge g$ where f and g are Boolean functions with $var(f) \neq \emptyset$, $var(g) \neq \emptyset$, and $var(f) \cup var(g) = \emptyset$.

▷ Claim 16. Let G be a connected graph whose vertices are seen as variables. The formula $F_G = \bigwedge_{\{x,y\} \in G} x \vee y$ cannot be decomposed. In particular, its backbone is empty.

Proof. Consider a partition (X_1, X_2) of $var(F_G) = V(G)$ where X_1 and X_2 are non-empty. We prove that $F_G \not\equiv f(X_1) \wedge g(X_2)$. We have an edge (x_1, x_2) of G such that $x_1 \in X_1$ and $x_2 \in X_2$. If $F_G \equiv f \wedge g$ then we cannot have that f has a model where x_1 is set to 0 and that at the same time g has a model where x_2 is set to 0, for otherwise $f \wedge g$ would have a model that falsifies $x_1 \vee x_2$. Since F_G has models where x_1 is set to 0 and others where x_2 is set to 0, F_G is not equivalent to $f(X_1) \wedge g(X_2)$. ◁

As explained before, we use the path p from Lemma 14 and we assume, without loss of generality, that $|b(p) \cap X| \geq n/3$. Let H be the spine subgraph given by Lemma 12 for the X-grid and $S = b(p) \cap X$. We call S' the vertices of S that see $spine(H)$ despite S. By Lemma 12, we have $|S'| = \Omega(\sqrt{n})$. Let $V \subseteq X$ be the set of vertices of the whole grid that see $spine(H)$ despite S and let $\bar{V} = X \setminus V$. It holds that $spine(H) \cup S' \subseteq V$. For instance, if H is the spine subgraph represented in Figure (5a) with the S-vertices circled, then the black vertices are V. $H[V]$ is then the subgraph represented in Figure (5b).

Let C be a decision-DNNF circuit organized by $\mathcal{T}$ and let β be the assignment to $Z \cup X' \cup \bar{V}$ that sets all variables of $\bar{V} \cup X'$ to 1 and that sets Z to select the subgraph H. By Lemma 7, $C' = C|\beta$ is a decision-DNNF circuit computing $F_n|\beta = F_H|\beta = F_{H-\bar{V}} = F_{H[V]} = \bigwedge_{\{x,y\} \in E(H[V])}(x \vee y)$, and that is organized by $\mathcal{T}' = (T, V \cap b)$. We write $b' = V \cap b$. We prove several intermediate claims.

For t a node in p, p_t is the subpath of p from the root node of T to t. Recall that $b(p) = \bigcup_{t \in p} b(t)$.

▷ Claim 17. Let t be a node of p such that $b'(p_t) \subseteq S'$. If t has a child c not in p such that $b'_{\downarrow}(c) \nsubseteq S'$, then for all $x \in S'$, $t \leq_T t_x$.

Proof. $S' \subseteq b(p)$ so, for all $x \in S'$, $t_x \in p$. Suppose there exists $x \in S'$ and $t \in p$ such that $t_x <_T t$ and t has a child $c \notin p$ with $b'_{\downarrow}(c) \nsubseteq S'$. We have that $c \notin p$. Let $y \in b'_{\downarrow}(c) \setminus S'$. Then $x \notin b'(t)$ and, since $b'(t) \subseteq b'(p_t) \subseteq S'$, we have $y \notin b'(t)$. So there are variables not in $b'(t)$ appearing in bags under two distinct children of t and thus $b'(t)$ is a separator of $H[V]$ such that $H[V] - b'(t)$ has two non-empty components. But that cannot be, because $b'(t) \subseteq S'$ and neither S' nor any of its subset is a separator of $V[H]$. ◁

p contains the root r of T and $b(p) \cap X = S$, so $b'(r) \subseteq S'$ holds. So by Claim 17, on any path from C''s root to a sink, the variables of S' appear first: every path q from C''s source to the 1-sink, can be written $q' \cdot q''$ where only variables of S' appear in q' and where the q'' contains no variable of S'.

There can be $\wedge$-nodes in q'. But there are particular $\wedge$-nodes. A decomposable $\wedge$-node is called an $\wedge^*$-node if exactly one of its children does not compute a term, i.e., a conjunction of literals. The child in question is called the *non-term part* of the node, and the term obtained as the conjunction of all terms under all other children is called the *term part* of the node. For instance, Figure 2 represents a $\wedge^*$-node: the first three children all represent terms but the fourth one (on the right) does not. We are going to prove that every $\wedge$-node appearing in q' is $\wedge^*$-node

For $v \in S'$, we denote by $\eta(v)$ its unique neighbor in $H[V]$ and for any given $S'' \subseteq S'$ we write $\eta(S'') = \{\eta(v) \mid v \in S''\}$. See for instance Figure (5b).

▷ Claim 18. Let α be a partial assignment to S'. $F_{H[V]}|\alpha$ is equivalent to $\tau_\alpha \wedge F_{H_\alpha}$ where $\tau_\alpha = \bigwedge_{\alpha(v)=0} \eta(v)$ and $H_\alpha = H[V] - (var(\alpha) \cup var(\tau))$ is connected, and F_{H_α} cannot be decomposed.

Proof. The clause $v \vee \eta(v)$ forces that if $\alpha(v) = 0$, all models of $F_{H[V]}|\alpha$ must assign $\eta(v)$ to 1. For convienence, see τ_α as the assignment $\eta(v)$ that sets $\eta(v)$ tp 1 for all v such that $\alpha(v) = 0$. Then $F_{H[V]}|\alpha = \tau_\alpha \wedge F_{H[V]}|\tau_\alpha = \tau_\alpha \wedge F_{H[V]-var(\alpha)-var(\tau_\alpha)} = \tau_\alpha \wedge F_{H_\alpha}$. Since all vertices of S' are at distance 2 from $spine(H)$, we have that $H[V] - var(\alpha) - var(\tau_\alpha)$ is connected. So by Claim 16, F_{H_α} is not decomposable. ◁

▷ Claim 19. Let $q = (v_1, \dots, v_m) = q' \cdot q''$ be a path in C' with only variables of S' appearing in q' and no variables of S' appearing in q''. Let γ be the (partial) assignment to S' corresponding to q'. Then every $\wedge$-node v_l in q' is a $\wedge^*$-node whose term part is a subterm of $\tau_\gamma = \bigwedge_{\gamma(v)=0} \eta(v)$.

Proof. Let v_k be the first $\wedge$-node in q' and let α be the assignment corresponding to $(v_1, \dots, v_k)$. By Claim 18, v_k is a $\wedge^*$-node whose term part is $\tau_\alpha = \bigwedge_{\alpha(v)=0} \eta(v)$ or a subterm of τ_α, which is itself a subterm of τ_γ.

Now suppose v_ℓ is some $\wedge$-node in q', let again α be the assignment corresponding to $(v_1, \dots, v_\ell)$ and suppose all $\wedge$-nodes before v_ℓ on that path are $\wedge^*$-nodes whose term parts are subterms of τ_α. By Claim 18, $F_{V[H]}|\alpha$ is equivalent to $F_{H_\alpha} \wedge \tau_\alpha$ where F_{H_α} is not decomposable, so C'_{v_ℓ} computes $F_{H_\alpha} \wedge \tau'$ where τ' is τ_α minus all term parts of all $\wedge^*$-nodes before v_ℓ. So v_ℓ is a $\wedge^*$-node whose term part is τ' or a subterm of τ', and therefore a subterm of τ_γ. ◁

$F_{H[V]}|\gamma$ is satisfiable for every *complete* assignment γ to S' so, for every γ, we can construct a path q as follows: starting from v_1, if the current node v is a decision node for $x \in S'$, follows the $\gamma(x)$-child of v, if instead v is a $\wedge^*$-node then follow the non-term child, and otherwise adds v to q and stops. This path is unique. We call it q_γ. We claim that q_γ is not missing any variable of S'

▷ Claim 20. For every complete assignment γ to S', q_γ contains one decision node for every variable in S'.

Proof. Let v be the last node of q_γ. Let α be the variable assignment to S' corresponding to q_γ. Clearly α is consistent with γ, but we have to show that $var(\alpha) = var(\gamma)$. By Claim 18, $F_{H[V]}|\alpha$ is equivalent to $F_{H_\alpha} \wedge \tau_\alpha$ where $H_\alpha = H[V] - (var(\alpha) \cup var(\tau_\alpha))$. By Claim 19, the term part of the $\wedge^*$-nodes in q are subterms of τ_α. So C'_v represents a function $F_{H_\alpha} \wedge \tau'$ for τ' a subterm of τ_α (τ' is possibly empty). C'_v has models and counter models so v is not a sink, it also not a $\wedge^*$-node by construction of q_γ, nor is it a more general decomposable $\wedge$-node because F_{H_α} is not decomposable. So v is a decision node for a variable y, and $y \notin S'$ by construction of q_γ. If α is not a complete assignment of S', then H_α contains a variable/vertex x of S'. x is essential in F_{H_α} so it must appear in C'_{v_m}. Let y be the variable for v_m. Since $y \notin S'$, we deduce from Claims 17 that $t_y <_T t_x$. So x cannot exist for otherwise C' would not be organized by $\mathcal{T}'$. ◁

Now, the circuit C'_v rooted under the last node v of q_γ computes $F_{H_\gamma} \wedge \tau$, where τ is τ_γ or a subterm of τ_γ. We can then show that the functions computed are logically distinct for distinct γ.

▷ Claim 21. Let γ and γ' be distinct complete assignments to S' and let v and v' be the last nodes of q_γ and q'_γ, respectively, then C'_v and $C'_{v'}$ are not logically equivalent

Proof. Let $x \in S'$ be a variable where $\gamma(x) \neq \gamma'(x)$. Say $\gamma(x) = 1$. C'_v computes $F_{H'} \wedge \tau$ for τ a subterm of $\tau_\gamma = \bigwedge_{\gamma(v)=0} \eta(v)$ and $H' = H[V] - S' - var(\tau)$. Note that $\eta(x)$ is in H' so $F_{H'}$ and C'_v essentially depend on $\eta(x)$ and have models where $\eta(v)$ is set to 0. $C'_{v'}$ computes $F_{H''} \wedge \tau'$ for τ' a subterm of $\tau_{\gamma'} = \bigwedge_{\gamma'(v)=0} \eta(v)$ and $H'' = H[V] - S' - var(\tau')$. Either $\eta(x)$ is in τ' and then $C'_{v'} \models \eta(x)$, or $\eta(x)$ is not in τ' and then $C'_{v'}$ does not depend on $\eta(x)$. In both cases, C'_v and $C'_{v'}$ are not logically equivalent. ◁

So we have $2^{|S'|}$ paths (one per γ) each containing a unique node. So C' contains at least $2^{|S'|} = 2^{\Omega(\sqrt{n})}$ nodes and this finishes the proof of Lemma 15.

6 Hard Functions for Dynamic Top-Down Compilation

We show the opposite variant of Theorem 8.

▶ **Theorem 22.** *There is an infinite class $\mathcal{F}$ of CNF formulas and a constant $\delta \in (0,1]$ such that, for every $F \in \mathcal{F}$ over n variables we have $tw(G_F) = o(n)$ and*

- $\texttt{Compile}_{d,\epsilon}[smc, bb](F)$ *returns a decision-DNNF circuit of size $2^{\Omega(n^\delta)}$;*
- *there exists a tree decomposition $\mathcal{T}$ of width $O(tw(G_F))$ such that $\texttt{Compile}_s(F, \mathcal{T})$ returns a decision-DNNF circuit of size $n^{O(1)}$.*

There is an asymmetry compared to Theorem 8 though: Theorem 22 is a positive result for the static compilation approach for *some* well-chosen tree decomposition of close-to-minimal width. The result does not hold for all tree decompositions of with the same width.

For the proof we will have $\delta = 1/4$. We have not looked to optimize this exponent and we could probably do better with formulas more cleverly crafted (but probably more complex). The point here is just to show that one algorithm has a polynomial size output while the other does not.

It is counter intuitive that, in our settings where many aspects of the algorithms are non-deterministic, the dynamic approach can be outperformed by its static counterpart. The idea is to design formulas where there is a clear optimal order to evaluate variables that can be hinted to the static approach via the tree decomposition, while the dynamic approach cannot take advantage of it. The hard formulas are of the form

$$F_n(X, X', X'') = L_n(X, X') \wedge R_n(X, X'')$$

L_n and R_n only share the X-variables. Roughly put, L_n's role is to ensure that the first cutset selected by $\texttt{Compile}_{d,\epsilon}$ contains X, and R_n is a formula that is hard to compile when the X-variables are the first selected to be assigned. There are many possibilities for L_n and R_n. Let us start with the formula chosen for R_n.

$$R_n(X, Y, Z) = \bigwedge_{i \in [0, n-1]} \bigwedge_{j \in [n]} (\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}) \wedge (\bar{z}_1 \vee \cdots \vee \bar{z}_{n^2})$$

with $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_{n^2}\}$ and $Z = \{z_1, \ldots, z_{n^2}\}$ (so $X'' = Y \cup Z$).

▶ **Lemma 23.** *Let S such that $X \subseteq S$ and $|S| = O(n)$. The decision-DNNF circuit returned by $\texttt{Compile}_{d,\epsilon}[smc, bb](R_n, S)$ has size at least $2^{\Omega(n)}$.*

Proof. We call a clause $\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}$ a small clause and $\bar{z}_1 \vee \cdots \vee \bar{z}_{n^2}$ the big clause. Since $|S| = O(n)$, we have that $|(Y \cup Z) \setminus S| = \Omega(n^2)$. Now let $I \subseteq [n]$ such that $i \in I$ if and only if $z_{j+in} \notin S$ and $y_{j+in} \notin S$ for some $j \in [n]$. Since $|S| = O(n)$, we have that $|I| = \Omega(n)$.

Let β be the assignment that maps all variables in $(Y \cup Z) \cap S$ to 1 and all x_i for $i \notin I$ to 0. For α a partial assignment to S consistent with β, the graph of $R_n|\alpha$ remains connected because α does not satisfies the big clause. The small clauses that remain in $R_n|\alpha$ are of the form $\bar{x}_{i+1} \vee y_{j+in} \vee z_{j+in}$ or $y_{j+in} \vee z_{j+in}$. We claim that the backbone of $R_n|\alpha$ is empty. To show this, it is sufficient to describe two families of assignments that satisfy $R_n|\alpha$:

- all assignments that set all Y-variables to 1 and one remaining Z-variable to 0 satisfy $R_n|\alpha$;
- for any $y_j \in var(R_n|\alpha)$, z_j must be in $var(R_n|\alpha)$ (because they appear together in a small clause), and there must be $z_{j'} \in var(R_n|\alpha) \setminus \{z_j\}$ (because S is too small to leave $R_n|\alpha$ with only one Z-variable) so assigning y_j to 0, z_j to 1, $z_{j'}$ to 0 and all remaining variables to 1 satisfies $R_n|\alpha$.

So, in each branch followed by $\texttt{Compile}_{d,\epsilon}[smc, bb](R_n, S)$ that leads to the residual formula $R_n|\alpha$ for some α consistent with β, no decomposable $\wedge$-nodes are created because the graph always stays connected, and `process` has no effect because the backbone is always empty. So for each *complete* assignment α to S consistent with β, the corresponding branch of the algorithm creates only decision nodes. For any two distinct *complete* assignments α and α' to S consistent with β, if α and α' disagree on a variable, then it is some X-variable x_i for $i \in I$ and we have that $R_n|\alpha \not\equiv R_n|\alpha'$ because one formula essentially depends on some $y_{j+in} \notin S$ while the other does not. So even with semantical caching, the circuit contains at least one distinct node per α, so at least $2^{|I|} = 2^{\Omega(n)}$ nodes. ◀

Now for L_n consider two disjoint $n^2 \times n^2$ grids G_1 and G_2 with vertices $V = \{v_{ij} \mid i,j \in [n^2]\}$ and $U = \{u_{ij} \mid i,j \in [n^2]\}$, and the two formulas $F_{G_1}(V) = \bigwedge_{(v,v') \in E(G_1)} (v \Leftrightarrow v')$ and $F_{G_1}(U) = \bigwedge_{(u,u') \in E(G_2)} (u \Leftrightarrow u')$. Then

$$L_n(X,U,V) = \bigwedge_{c \in F_{G_1} \wedge F_{G_2}} c \vee x_1 \vee \cdots \vee x_n \equiv (F_{G_1} \wedge F_{G_2}) \vee x_1 \vee \cdots \vee x_n$$

F_{G_1} and F_{G_2} are trivial. Their only models are the assignments where all V-variables and all U-variables are set to 0 or 1. Arguably we could have designed less trivial formulas. Most formulas with a grid-like primal graphs that are easy to compile would be acceptable substitutes. We are only interested in their primal graphs for the following lemma.

▶ **Lemma 24.** *For n large enough, every $2/3$-balanced separator S of $G_{L_n \wedge R_n}$ of size $O(n)$ contains X.*

Proof. Suppose S does not contain X in its entirety. By Lemma 9, there are no $3/4$-balanced separator of G_1 or G_2 of size $O(n)$. Thus both $G_{F_1} - S$ and $G_{F_2} - S$ contain a connected component of size at least $3n^4/4$. Let H_1 and H_2 be these components. But then H_1 and H_2 are connected to a vertex $x \in X \setminus S$ in $G_{L_n \wedge R_n} - S$ and thus $G_{L_n \wedge R_n} - S$ has a component of size at least $6n^4/4$. The number of variables of $L_n \wedge R_n$ is $2n^4 + O(n^2)$ and $6n^4/4 > 4n^4/3 + O(n^2)$ for n large enough. So S is not a 2/3 balanced separator of $L_n \wedge R_n$. ◀

It is readily verified that the backbone of $L_n \wedge R_n$ is empty and that its graph is connected, so the first task of $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ is to find a balanced separator S of minimal size up to a factor $(1+\epsilon)$. Thus $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns the same circuit as $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n, S)$. By Lemma 24, the chosen separator S contains X and has size $O(n)$. If $S \subseteq X \cup Y \cup Z$ then by Lemma 23 we have that $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns a circuit of size $2^{\Omega(n)}$. Even if S contains some U- and V-variables, we can show that the same lower bounds hold. This is because assigning U- and V-variables cannot disconnect the primal graph as long as some X-variables are left unassigned and the impact of assigning such variables on the backbone of the formula is quasi-null.

▶ **Lemma 25.** *$\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n)$ returns a circuit of size $2^{\Omega(n)}$.*

Proof sketch. By Lemma 24, $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n)$ selects a separator S of size $O(n)$ that contains X. S may contain some U- and V-variables but, once an X-variable is set to 1, all clauses of L_n disappear and all U- and V-variables with them. $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n)$ returns the same circuit as $\texttt{Compile}_{d,\epsilon}(L_n \wedge R_n, S)$.

We follow the proof of Lemma 23. The set I and the assignment β are defined the same way. For α a partial assignment to S consistent with β and that either assigns an X-variable to 1 or does not assign two X-variables, the graph of $(L_n \wedge R_n)|\alpha$ remains connected. Indeed α does not satisfies the big clause of R_n because it is consistent with β, and if it assigns an X-variable to 1 then the part of the graph for L_n vanishes, otherwise if some X-variable is left unassigned then L_n's graph and R_n's graph remains connected through that variable.

We claim that the backbone of $(L_n \wedge R_n)|\alpha$ is empty. If α sets an X-variable to 1 then $(L_n \wedge R_n)|\alpha = R_n|\alpha$ so we can just use the proof of Lemma 23. Otherwise, α sets all its X-variables to 0 but let at least 2 unassigned. In this case, clauses of $L_n|\alpha$ still exist and contain two X-variables x and x'. We can then extend the assignments to the Y-variables and the Z-variables described in the proof of Lemma 23 by setting one of x or x' to 0 and the other to 1. Such assignments satisfy $L_n \wedge R_n|\alpha$ and suffice to show that no X-, Y-, Z-, U- or V- literals can be in the backbone of $L_n \wedge R_n|\alpha$.

So, each branch followed by $\texttt{Compile}_{d,\epsilon}[smc, bb](L_n \wedge R_n, S)$ for one of the assignment α described above leads to the residual formula $(L_n \wedge R_n)|\alpha$ without creating a single decomposable $\wedge$-node nor benefiting from `process`. In particular, along each branch for each a *complete* assignment α to S' that is consistent with β and assigns at least one X-variable to 1, the algorithm creates only decision nodes. Since for any two such complete assignments α and α' we have that $(L_n \wedge R_n)|\alpha = R_n|\alpha$ and $(L_n \wedge R_n)|\alpha' = R_n|\alpha'$, and since we know by the proof of Lemma 23 that $R_n|\alpha \not\equiv R_n|\alpha'$, we deduce that each branch for these α contains a unique node. So there is are least $2^{|I|} - 1 = 2^{\Omega(n)}$ nodes (-1 because we remove the assignment where all X-variables are set to 0). ◀

It remains to study $\texttt{Compile}_s$ on $L_n \wedge R_n$. There is a tree decomposition we can use to force $\texttt{Compile}_s$ to assign the variables in the right order π, namely:

$$\begin{aligned}\pi : {} & x_1, y_1, z_1, y_2, z_2, y_3, z_3, \dots, y_n, z_n,\\ & x_2, y_{1+n}, z_{1+n}, y_{2+n}, z_{2+n}, y_{3+n}, z_{3+n}, \dots, y_{2n}, z_{2n}, \dots\\ & x_n, y_{1-n+n^2}, z_{1-n+n^2}, y_{2-n+n^2}, z_{2-n+n^2}, \dots, y_{n^2}, z_{n^2}, \dots\end{aligned}$$

where the U-variables and the V-variables are put at the end in any order. This is basically the reading order of the X, Y, Z-variables as they appear in R_n. To force $\texttt{Compile}_s$ to read the variable in that order, it suffices to use a *nice* path decomposition, that is, a tree decomposition $\mathcal{T} = (T, b)$ where T is a path and where, first the root bag is empty and, second, for every two consecutive nodes t and t' in T, $b(t) = b(t') \cup \{x\}$ or $b(t) = b(t') \setminus \{x\}$ for some variable x. In a nice path decomposition, the function $x \mapsto t_x$ is injective (recall that t_x is the highest node of T whose bag contains x): there is a total order σ in which the variables appear in the bags of $\mathcal{T}$. So a decision-DNNF circuit organized by a nice path decomposition is guaranteed that on all its paths, the ordering of the variables for the decision nodes is consistent with σ. We just have to make sure that $\sigma = \pi$.

We use a nice path decomposition where, in a nutshell, the X, Y, Z-variables are added to the bags in the order given by π until we obtain one big bag containing $X \cup Y \cup Z$, then $Y \cup Z$ are removed and the remaining is an $O(n^2)$-width path decomposition of G_1 with X added to all bags, followed by an $O(n^2)$-width path decomposition of G_2 with X is added to all bags. The treewidth of $L_n \wedge R_n$ is $\Theta(n^2)$ because of the $n^2 \times n^2$ grids in L_n, so the path decomposition has width $O(tw(G_{L_n \wedge R_n}))$.

▶ **Lemma 26.** *There exists a tree decomposition $\mathcal{T}$ of width $O(tw(G_{L_n \wedge R_n}))$ such that* $\texttt{Compile}_s(L_n \wedge R_n, \mathcal{T})$ *returns a circuit of size $n^{O(1)}$.*

Proof sketch. In the circuit returned by $\texttt{Compile}_s(L_n \wedge R_n, \mathcal{T})$, the decision nodes follow the order π. The clauses $(\bar{x}_i \vee y_{j+in} \vee z_{j+in}$ are then falsified or satisfied in increasing order of i and j. For every assignment α to $\{x_i, y_{j+in} \mid i < k \text{ and } j + in \leq h\}$, only a constant number of formulas $(L_n \wedge R_n)|\alpha$ are possible, so syntactical caching ensures that there are only a constant number of decision nodes labeled by the same variable in the circuit. ◀

Theorem 22 follows from the combination of Lemmas 25 and 26.

7 Conclusion

We have studied the relative efficiency of two top-down compilation algorithms. Both use variable selection mechanisms inspired from practical compilers. They both select batches of variables that correspond to separators of the primal graph, but one uses a dynamic graph balanced-partitioning approach while the other relies on a pre-computed tree decomposition. We have shown that the two algorithms construct large decision-DNNF circuits on instances that yet admit polynomial-size decision-DNNF circuits. Moreover we have shown that there are instances where only one of two approaches fails to construct a small circuit. Often in knowledge compilation, we compare languages saying that they offer small compiled forms for different kind of functions. What we show here is that even within the same language, similar comparisons are possible for the compilation algorithms: the classes of formulas for which two compilers to decision-DNNF are able to find polynomial-size compiled forms can be distinct and not included in one another. This calls for criterion for deciding to which compilers an instance should be sent.

References

1 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #sat and bayesian inference. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 340–351. IEEE Computer Society, 2003. `doi:10.1109/SFCS.2003.1238208`.

2 Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Lower bounds for exact model counting and applications in probabilistic databases. In Ann E. Nicholson and Padhraic Smyth, editors, *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013.* AUAI Press, 2013. URL: `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2366&proceeding_id=29`.

3 Béla Bollobás and Imre Leader. Edge-isoperimetric inequalities in the grid. *Comb.*, 11(4):299–314, 1991. `doi:10.1007/BF01275667`.

4 Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf. Preprocessing of intractable problems. *Inf. Comput.*, 176(2):89–120, 2002. `doi:10.1006/INCO.2001.3043`.

5 Andrea Calì, Florent Capelli, and Igor Razgon. Non-fpt lower bounds for structural restrictions of decision DNNF. *CoRR*, abs/1708.07767, 2017. `arXiv:1708.07767`.

6 Hubie Chen. Parameterized compilability. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 412–417. Professional Book Center, 2005. URL: `http://ijcai.org/Proceedings/05/Papers/0644.pdf`.

7 Adnan Darwiche. Recursive conditioning. *Artif. Intell.*, 126(1-2):5–41, 2001. `doi:10.1016/S0004-3702(00)00069-2`.

8 Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In Rina Dechter, Michael J. Kearns, and Richard S. Sutton, editors, *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*, pages 627–634. AAAI Press / The MIT Press, 2002. URL: `http://www.aaai.org/Library/AAAI/2002/aaai02-094.php`.

9 Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 328–332. IOS Press, 2004.

10 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002. `doi:10.1613/JAIR.989`.

11 Hélène Fargier, Pierre Marquis, Alexandre Niveau, and Nicolas Schmidt. A knowledge compilation map for ordered real-valued decision diagrams. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 1049–1055. AAAI Press, 2014. `doi:10.1609/AAAI.V28I1.8853`.

12 Hélène Fargier and Jérôme Mengin. A knowledge compilation map for conditional preference statements-based languages. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 492–500. ACM, 2021. `doi:10.5555/3463952.3464014`.

13 Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 156–162. Professional Book Center, 2005. URL: `http://ijcai.org/Proceedings/05/Papers/0876.pdf`.

14 Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 157–162. AAAI Press / The MIT Press, 2000. URL: `http://www.aaai.org/Library/AAAI/2000/aaai00-024.php`.

15 Rafael Kiesel and Thomas Eiter. Knowledge compilation and more with sharpsat-td. In Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner, editors, *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, pages 406–416, 2023. `doi:10.24963/KR.2023/40`.

16 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 8:1–8:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.CP.2021.8`.

17 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. `doi:10.24963/IJCAI.2017/93`.

18 Jean-Marie Lagniez, Pierre Marquis, and Anastasia Paparrizou. Defining and evaluating heuristics for the compilation of constraint networks. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2017. `doi:10.1007/978-3-319-66158-2_12`.

19 Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. DSHARP: Fast d-DNNF Compilation with sharpSAT. In *Canadian Conference on Artificial Intelligence*, 2012.

20 Umut Oztok and Adnan Darwiche. On compiling CNF into decision-dnnf. 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 42–57. Springer, 2014. `doi:10.1007/978-3-319-10428-7_7`.

21 Cédric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 525–529. IOS Press, 2008. `doi:10.3233/978-1-58603-891-5-525`.

22 Knot Pipatsrisawat and Adnan Darwiche. New compilation languages based on structured decomposability. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 517–522. AAAI Press, 2008. URL: `http://www.aaai.org/Library/AAAI/2008/aaai08-082.php`.

23 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL: `http://www.satisfiability.org/SAT04/programme/21.pdf`.

SAT Encoding of Partial Ordering Models for Graph Coloring Problems

Daniel Faber
Department of Computer Science, University of Bonn, Germany

Adalat Jabrayilov
Adesso SE, Dortmund, Germany

Petra Mutzel
Department of Computer Science, University of Bonn, Germany

Abstract

In this paper, we revisit SAT encodings of the partial-ordering based ILP model for the graph coloring problem (GCP) and suggest a generalization for the bandwidth coloring problem (BCP). The GCP asks for the minimum number of colors that can be assigned to the vertices of a given graph such that each two adjacent vertices get different colors. The BCP is a generalization, where each edge has a weight that enforces a minimal "distance" between the assigned colors, and the goal is to minimize the "largest" color used.

For the widely studied GCP, we experimentally compare the partial-ordering based SAT encoding to the state-of-the-art approaches on the DIMACS benchmark set. Our evaluation confirms that this SAT encoding is effective for sparse graphs and even outperforms the state-of-the-art on some DIMACS instances.

For the BCP, our theoretical analysis shows that the partial-ordering based SAT and ILP formulations have an asymptotically smaller size than that of the classical assignment-based model. Our practical evaluation confirms not only a dominance compared to the assignment-based encodings but also to the state-of-the-art approaches on a set of benchmark instances. Up to our knowledge, we have solved several open instances of the BCP from the literature for the first time.

2012 ACM Subject Classification Theory of computation → Logic; Mathematics of computing → Graph coloring

Keywords and phrases Graph coloring, bandwidth coloring, SAT encodings, ILP formulations

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.12

Related Version *Full Version*: `https://arxiv.org/abs/2403.15961`

Supplementary Material *Software (Source Code)*: `https://github.com/s6dafabe/popsatgcpbcp`

Funding *Daniel Faber*: funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 459420781.
Petra Mutzel: This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project no. 459420781 and under Germany's Excellence Strategy–EXC-2047/1–390685813, and by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as part of the Lamarr-Institute for Machine Learning and Artificial Intelligence, LAMARR22B.

Acknowledgements We are grateful for the valuable remarks of the referees, which helped to improve the paper.

1 Introduction

The graph coloring problem (GCP) asks for assigning a set of positive integers, called *colors*, to the vertices of a graph such that no two adjacent vertices have the same color while minimizing the number of colors used. The problem has numerous applications, e.g. in register allocation [3], scheduling [15], and computing sparse Jacobian matrices [6]. For this reason, this problem has been the subject of a vast amount of literature (see e.g., [16][12] for surveys). However, finding an optimal coloring is known to be NP-hard, and compared to other NP-hard problems, like the travelling salesman problem or the knapsack problem, only relatively small instances can be solved to optimality. A generalization of the graph coloring problem is the *bandwidth coloring problem* (BCP). In this problem, every edge $\{u, v\}$ in the graph has an additional weight $d(\{u, v\})$, and for a coloring to be valid, the difference of the colors $c(u)$ and $c(v)$ must be at least $d(\{u, v\})$ (i.e. $|c(u) - c(v)| \geq d(\{u, v\})$. The goal is to minimize the largest used color. Note that for uniform edge distances $d(e) = 1$ for all edges $e \in E$, the BCP reduces to the GCP. The problem has applications in frequency assignment [5], where transmitters close to each other need to be assigned to sufficiently differing frequencies to prevent interference.

In this paper, we concentrate on exact approaches for solving the above mentioned problems GCP and BCP, in particularly SAT approaches as well as integer linear programming (ILP) approaches, which are both state-of-the-art for solving coloring problems on graphs (see, e.g., [11, 14, 9, 8]).

SAT approaches are based on encoding the problem as a Boolean Satisfiability problem. A possible encoding consists of introducing color variables $x_{v,i}$, where a true assignment of $x_{v,i}$ represents assigning vertex v with color i (e.g., [11, 4]). Other methods are based on *Zykov's tree* induced by Zykov's deletion-contration recurrence (e.g., [9, 8]), in which the models contain variables $s_{u,v}$ that encode if vertices u and v have the same or different colors. Heule, Karahalios and van Hoeve [11] have introduced the algorithm *CliColCom* in which they alternatingly solve a maximum clique problem and a graph coloring problem using SAT approaches, where the solution from one problem helps finding a solution for the other problem and vice versa. Most relevant to this work are the SAT encodings suggested by Tamura et al. [21] and Ansótegui et al. [1], which contains binary variables $y_{v,i}$ for every vertex v and possible color i, indicating if color i is smaller than vertex v. Although the experimental evaluation in [21] has shown that this encoding has dominated the assignment SAT encoding, these results have caught little attention in the recent literature.

The most natural ILP model is the *assignment-based* ILP model, which directly assigns colors to vertices by introducing binary variables $x_{v,i}$ that (similar to the color variables in SAT) decide if vertex v is assigned to color i. A drawback of this formulation is the presence of symmetries in the solution space: For a valid coloring, any permutation of the color labels provides another equivalent solution leading to a significant larger search space. Mendez-Diaz and Zabala [19] have suggested additional symmetry-breaking constraints, that completely eliminate this type of symmetry. Mutzel and Jabrayilov [13] have proposed ILP formulations, which are based on formulating the coloring problem as a partial-ordering problem (POP). This model suggests ordering the colors and placing the vertices relatively in this order, analogous to the ordering encoding in [21] and [1] for SAT. It has been shown that for sparse graphs, the simple assignment and partial-ordering based models show good performance [13] and that the partial-ordering based models dominate the assignment-based models. Furthermore, [14] have shown theoretical advantages of POP over the assignment ILP. A rather complex but also competitive ILP model is based on the *set covering* formulation [18] (see, e.g. [10]), which uses the fact that a coloring describes a partitioning of the vertices into independent sets, and contains a variable x_s for every independent set s in the graph, that decides if s is a set of vertices corresponding to a color class in the coloring. Because there can be an exponential amount of independent sets, the formulation cannot be solved using standard techniques and instead has to be solved using column generation methods.

For the bandwidth coloring problem, there mainly exist numerous heuristic algorithms [17]. Two exact approaches are presented in [5]. The first approach uses a constraint programming formulation, which contains $|V|$ variables $x(v) \in [1, H]$ for $v \in V$ and $|E|$ constraints $|x(u) - x(v)| \geq d(\{u, v\})$ for every $\{u, v\} \in E$. The second one is based on the assignment-based ILP model, which contains constraints for every edge $\{u, v\}$ and every pair of colors i, j having a smaller difference than $d(\{u, v\})$. A drawback of this model is the high number of constraints, which depends on the size of the edge weights. We are not aware of any other exact approaches for the bandwidth coloring problem in the literature.

Our contribution. Motivated by the recent interest of the ILP community in partial-ordering based ILP models, we revisited SAT encodings of the partial-ordering based model for the GCP and generalize them to the BCP. For the GCP, we also strengthen the model using the symmetry-breaking constraints by Mendez-Diaz and Zabala [19] in order to eliminate the inherent symmetries in the solution space. Our experimental evaluation for the GCP shows that the partial-ordering based SAT encoding of the POP model outperforms the assignment-based SAT encoding as well as all evaluated ILP formulations from the literature on the DIMACS benchmark set.

Moreover, for the bandwidth coloring problem, we suggest a new modification of the partial-ordering based SAT and ILP models, which needs only one constraint per edge and color. Compared to the assignment based model for bandwidth coloring presented in [5], it has an asymptotically smaller number of constraints. This advantage of a more compact formulation size holds true for the SAT as well as the ILP formulations. Our computational experiments for the bandwidth coloring problem confirm that the new SAT encodings clearly outperform not only the classical assignment-based formulations but also the published state-of-the-art approaches. Our new SAT encodings solve much more instances to provable optimality within one hour of running time than the published approaches and have a significantly lower runtime on a large part of the instances.

2 State-of-the-art encodings

First, we present state-of-the-art encodings (models) that are relevant for our work. Subsequently, we discuss the state-of-the-art on exact solvers for the GCP and the BCP.

We use the following notation: For a graph $G = (V, E)$, we denote its vertex set by $V(G)$ and its edge set by $E(G)$. Each edge of an undirected graph is a 2-element subset $e = \{u, v\}$ of $V(G)$. The end vertices u, v of an edge $\{u, v\}$ are called adjacent vertices or neighbors. For given positive edge distances $d(u, v)$ for all $u, v \in V(G)$, we denote the average edge distance in G with $\bar{d}$. Each valid coloring partitions the vertices into independent sets, where each independent set corresponds to the set of vertices assigned to a specific color.

The formal definitions of the graph coloring variants studied in the paper are as follows. Given an undirected graph $G = (V, E)$, the *graph coloring problem* (GCP) asks for an assignment $c\colon V \to \mathbb{N}$ minimizing $\max_{v \in V} c(v)$, such that $c(u) \neq c(v)$ for all $\{u, v\} \in E$. Given an undirected graph $G = (V, E)$ and edge distances $d\colon E \to \mathbb{N}$, the *bandwidth coloring problem* (BCP) asks for an assignment $c\colon V \to \mathbb{N}$ satisfying $|c(u) - c(v)| \geq d(\{u, v\})$ for all $\{u, v\} \in E$, that minimizes $\max_{v \in V} c(v)$.

2.1 Integer programming formulations

In the next section, we discuss the ILP models that are relevant for this work. We use H to denote an arbitrary upper bound on the solutions of GCP and BCP, respectively. For example, a trivial upper bound for the GCP is $H = |V|$ and for the BCP is $H = |V| \cdot \max\{d_e : e \in E\}$.

2.1.1 The assignment models (`ASS-I`) and (`ASS-I-B`)

The classical ILP model for graph coloring is based on directly assigning a color $i = 1, ..., H$ to each of the vertices $v \in V$. For this it introduces binary variables $x_{v,i} \in \{0, 1\}$ for each $i = 1, ..., H$ and $v \in V$, which indicate if color i is assigned to vertex v (in this case $x_{v,i} = 1$, otherwise $x_{v,i} = 0$). To model the objective function, additional binary variables w_i for all colors $i = 1, ..., H$ are introduced, which indicate if a color i is used. This model is given by:

$$
\begin{array}{lll r}
\min & \displaystyle\sum_{i=1}^{H} w_i & & \\
\text{s.t.} & \sum_{i=1}^{H} x_{v,i} = 1 & \forall v \in V & \text{(1a)} \\
& x_{u,i} + x_{v,i} \leq w_i & \forall \{u, v\} \in E, i = 1, ..., H & \text{(1b)} \\
& w_i \leq \sum_{v \in V} x_{vi} & \forall i = 1, \ldots, H & \text{(1c)} \\
& w_i \leq w_{i-1} & \forall i = 2, \ldots, H & \text{(1d)} \\
& x_{v,i}, w_i \in \{0, 1\} & \forall v \in V, i = 1...H & \text{(1e)}
\end{array}
$$

Equation (1a) ensures that each vertex is colored with exactly one color. Equation (1b) guarantees that adjacent vertices have different colors and that variable w_i is set to 1 if a vertex is colored with i. Finally, the objective minimizes the number of used colors. A main drawback of the original model using (1a), (1b), and (1e) only is that there are $\binom{H}{\chi}$ possibilities to select χ from H colors. This results in many optimal solutions that are symmetric to each other. In order to overcome this symmetry, Mendez-Diaz and Zabala [19] have suggested to add the constraints (1c) and (1d).

The model contains additional symmetries that arise due to the arbitrary labeling of the colors: For every valid solution, one can obtain an equivalent solution by swapping the labels of two colors. Mendez-Diaz and Zabala [19] propose additional constraints to break these symmetries:

$$x_{v,i} = 0 \qquad \forall i > v, v \in 1, ..., H \tag{2a}$$

$$x_{v,i} \leq \sum_{u=i-1}^{v-1} x_{u,i-1} \qquad \forall v \in V \setminus \{1, |V|\}, i = 2, ..., H \tag{2b}$$

The assignment model, strengthened with the symmetry-breaking constraints, has the form

$$\texttt{ASS-I}: \quad \min \Big\{ \sum_{i=1}^{H} w_i \colon x, w \text{ satisfy (1a)–(1e), (2a)–(2b)} \Big\}.$$

Adaptation of the assignment model to the bandwidth coloring model

Dias et al. [5] suggested an extension to the assignment model to solve the bandwidth coloring problem. The idea is to modify the edge constraints (1b), such that for every edge e and every pair of colors i, j with $|i - j| < d(e)$, at most one of the two colors can be assigned to the two incident vertices. The full model presented in the paper is given below:

$$
\begin{array}{llll r}
\texttt{ASS-I-B}: & \min & z_{max} & & \\
& \text{s.t.} & \sum_{i=1}^{H} x_{v,i} = 1 & \forall v \in V & \text{(3a)} \\
& & x_{u,i} + x_{v,j} \leq 1 & \forall e = \{u, v\} \in E, & \\
& & & \forall i, j = 1, ..., H \text{ with } |i - j| < d(e) & \text{(3b)} \\
& & z_{max} \geq i \cdot x_{v,i} & \forall v \in V, i = 1, ..., H & \text{(3c)} \\
& & x_{v,i} \in \{0, 1\}, z_{\max} \in \mathbb{R} & \forall v \in V, i = 1, ..., H & \text{(3d)}
\end{array}
$$

To describe the largest used color, the formulation uses a continuous variable z_{max} instead of using H binary variables $w_1, ..., w_H$, since in an optimal solution of the BCP the largest assigned color can be greater than the number $\sum_i^H w_i$ of assigned colors. Constraints (3c) ensure that if there is a vertex with color i, then the largest used color z_{max} is at least i, i.e. there is no used color larger than z_{max}. The correctness of the model has been shown in [5], we analyze the size of the model in the following.

▶ **Lemma 1.** *ASS-I-B contains $H \cdot |V| + 1$ variables and $(H+1) \cdot |V| + H \cdot |E|(2\bar{d} - 1) - \sum_{e \in E} \left(d(e)^2 - d(e)\right)$ constraints.*

Proof. Obviously, the model contains $H \cdot |V| + 1$ variables and $(H+1) \cdot |V|$ constraints of type (3a) and (3c). The number of edge constraints in (3b) can be rewritten as

$$\begin{aligned}\sum_{e \in E} |\{(i,j) \in 1, ..., H \colon |i-j| < d(e)\}| &= \sum_{e \in E} \Big(H \cdot (2d(e) - 1) - (d(e)^2 - d(e))\Big) \\ &= 2H \sum_{e \in E} d(e) - H|E| - \sum_{e \in E} \left(d(e)^2 - d(e)\right) \\ &= H \cdot |E|(2\bar{d} - 1) - \sum_{e \in E} \left(d(e)^2 - d(e)\right).\end{aligned}$$

where $\bar{d}$ is the average edge distance in G. The first equality can be derived as follows: For every color i, the interval of colors j satisfying $|i-j| < d(e)$ is $j \in [i - d(e) + 1, i + d(e) - 1]$. This interval contains exactly $2d(e) - 1$ elements, which for the H colors $i = 1, ..., H$ leads to $H \cdot (2d(e) - 1)$ pairs (i, j) in total. However, we have to subtract the pairs we counted for which $j < 1$ or $j > H$. For every i with $i - d(e) < 1$, there exist exactly $d(e) - i$ pairs (i, j) for which $j < 1$: $(i, 0), (i, -1), ..., (i, i - d(e) + 1)$. In total, we have $\sum_{i=1}^{d(e)} (d(e) - i) = (d(e)^2 - d(e))/2$ of such pairs. For $j > H$ the situation is symmetrical, leading to a total number of $d(e)^2 - d(e)$ of pairs we need to subtract for each edge. ◀

2.1.2 The partial-ordering based model (POP-I) for the GCP

Jabrayilov and Mutzel [13] have suggested to interpret the coloring problem as a partial-ordering problem (POP). An advantage of this model is that it has less inherent symmetries between the colors than the assignment model. This model considers the colors $1, ..., H$ to be linearly ordered. Each vertex is then ordered relative to the colors, i.e., for each vertex its relative position with respect to the colors is determined. A color is then indirectly assigned to a vertex v if it is neither larger nor smaller than v. The variables $y_{v,i}$ for all $v \in V$ and $i = 1, ..., H$ indicate if color i is smaller than vertex v. In case i is smaller than v in the partial order (denoted by $v \succ i$), we have $y_{v,i} = 1$, otherwise $y_{v,i} = 0$. The color of a vertex is then the smallest color that is not smaller than v, i.e., the color i for which $y_{v,i-1} - y_{v,i} = 1$ or in the case $y_{v,1} = 0$ the color $i = 1$. The partial-ordering based model has the following form, where q is an isolated dummy vertex added to G:

$$\begin{array}{llll}
\min & 1 + \sum_{i=1}^{H} y_{q,i} & & \\
\text{s.t.} & y_{v,H} = 0 & \forall v \in V & (4a) \\
 & y_{v,i} - y_{v,i+1} \geq 0 & \forall v \in V, i = 1, ..., H-1 & (4b) \\
 & y_{u,1} + y_{v,1} \geq 1 & \forall \{u,v\} \in E & (4c) \\
 & y_{u,i-1} - y_{u,i} + y_{v,i-1} - y_{v,i} \leq 1 & \forall \{u,v\} \in E, i = 2, ..., H & (4d) \\
 & y_{q,i} - y_{v,i} \geq 0 & \forall v \in V, i = 1, ..., H-1 & (4e) \\
 & y_{v,i} \in \{0,1\} & \forall v \in V, i = 1, ..., H & (4f)
\end{array}$$

Constraints (4a)–(4c) ensure that each vertex receives exactly one color from $1, ..., H$. Every adjacent pair of vertices must receive different colors. This is guaranteed by constraints (4d). Constraints (4e) enforce that there is no vertex $v \in V$ with $v \succ q$, i.e., the dummy vertex q has the largest used color. The objective function minimizes the number of colors $\sum_{i=1}^{H} y_{q,i}$ smaller than q incremented by one for the color assigned to q.

The variables of the partial-ordering based model and those of the assignment model are related in the following way:

$$x_{v,1} = 1 - y_{v,1} \qquad \forall v \in V \tag{5a}$$

$$x_{v,i} = y_{v,i-1} - y_{v,i} \qquad \forall v \in V, i = 2, ..., H \tag{5b}$$

Using these equations, the symmetry-breaking constraints (2a)–(2b) can be modified for the partial-ordering based model:

$$(2a) \Rightarrow \qquad y_{v,v} = 0 \qquad \forall v \in 1, ..., H \tag{6a}$$

$$(2b) \Rightarrow \qquad y_{v,i} \leq \sum_{u=i-1}^{v-1} (y_{u,i-1} - y_{u,i}) \qquad \forall v \in V \setminus \{1, |V|\}, i = 2, ..., H \tag{6b}$$

The partial-ordering based model, strengthened with the symmetry-breaking constraints, has the form

$$\texttt{POP-I}: \quad \min \Big\{ 1 + \sum_{i=1}^{H} y_{q,i} \colon y \text{ satisfy (4a)–(4f), (6a)–(6b)} \Big\}.$$

Notice that in [13] the vertex q is chosen from V. However, this would cause a conflict with the symmetry-breaking constraints. To avoid the conflict we add q as a new isolated vertex.

2.1.3 The hybrid partial-ordering based model (`POPH-I`)

Jabrayilov and Mutzel [13] observed that for growing graph density, the constraint matrix of the model (`POP-I`) contains more nonzero elements than the (`ASS-I`) constraint matrix. This is due to constraints (4d), which are responsible for adjacent vertices having different colors and contain four nonzero coefficients instead of the three in the corresponding constraints (1b) in (`ASS-I`). To circumvent this problem, they suggest a hybrid model (`POPH-I`): In this model, they include the variables $x_{v,i} \in \{0, 1\}$ with the constraints (5a)-(5b) and substitute the constraints (4d) by:

$$x_{u,i} + x_{v,i} \leq 1 \qquad \forall e = \{u, v\} \in E, i = 1, ..., H \tag{7}$$

The hybrid model, strengthened with the symmetry-breaking constraints, has the form

$$\texttt{POPH-I}: \ \min \Big\{ 1 + \sum_{i=1}^{H} y_{q,i} \colon x, y \text{ satisfy (1e), (4a)–(4b), (4e)–(4f), (5a), (5b), (7), (6a), (2b)} \Big\}.$$

2.2 SAT encodings (`ASS-S`) and (`ASS-S-B`)

Similar to the ILP encoding, the assignment model for graph coloring can also be encoded as a Boolean Satisfiability Problem (SAT). Since SAT is a decision problem, we cannot directly optimize the number of used colors. To find the chromatic number of a graph, one therefore encodes the k-colorability problem (i.e., the problem of deciding if a given graph can be colored using k colors). The assignment constraints (1a) and (1b) are sufficient to model the k-colorability. These constraints can be encoded using the following clauses:

$$\bigvee_{i=1}^{k} x_{v,i} \qquad \forall v \in V \tag{8a}$$

$$\neg x_{u,i} \vee \neg x_{v,i} \qquad \forall \{u,v\} \in E,\ i = 1, ..., k \tag{8b}$$

$$x_{v,i} \in \{True, False\} \qquad \forall v \in V, i = 1, ..., k$$

To encode that each vertex is also assigned to at most one color, one possible encoding is the sequential encoding [20], where the idea is to build a count-and-compare hardware circuit and translate it into conjunctive normal form (CNF). This encoding adds $3k - 4$ clauses and $k - 1$ auxiliary variables $s_{v,i}, i = 1, ..., k - 1$ per vertex v:

$$\neg x_{v,i} \vee s_{v,i} \qquad \forall v \in V, i = 1, ..., k-1 \tag{9a}$$

$$\neg s_{v,i-1} \vee s_{v,i} \qquad \forall v \in V, i = 2, ..., k-1 \tag{9b}$$

$$\neg x_{v,i} \vee \neg s_{v,i-1} \qquad \forall v \in V, i = 2, ..., k-1 \tag{9c}$$

$$\neg x_{v,k} \vee s_{v,k-1} \qquad \forall v \in V \tag{9d}$$

We remark that enforcing each vertex to have at most one color is not strictly necessary, however, it may improve performance as it eliminates redundant solutions from the search space. In our initial experiments, only enforcing each vertex to have at least one color or using the standard binomial encoding for the at-least-1 constraints showed subpar performance.

Translating the symmetry-breaking constraints (2a)-(2b) adds the following clauses:

$$\neg x_{v,i} \qquad \forall i > v, v \in 1, ..., k \tag{10a}$$

$$\neg x_{v,i} \vee \bigvee_{u=i-1}^{v-1} x_{u,i-1} \qquad \forall v \in V \setminus \{1, |V|\}, i = 2, ..., k \tag{10b}$$

similar symmetry breaking was also used in [11, 23].

The SAT encoding of the assignment model, strengthened with these symmetry-breaking constraints, has the following form:

`ASS-S`: consists of clauses (8a), (8b), (9a)–(9d), (10a), and (10b).

Adaptation to the bandwidth coloring model

To extend the previous SAT formulation into a formulation for the bandwidth coloring problem, one can modify the edge clauses (8b) analogous to the ILP model (`ASS-I-B`). The new edge clauses are:

$$\neg x_{u,i} \vee \neg x_{v,j} \qquad \forall e = \{u,v\} \in E, \forall i,j = 1, ..., H \text{ with } |i-j| < d(e) \tag{11}$$

The SAT encoding of the assignment model for the BCP has the following form:

`ASS-S-B`: consists of clauses (8a), (9a)–(9d), and (11).

3 Formulations based on the partial-ordering approach

Here, we revisit the SAT encoding suggested by [21] and [1] for the GCP which also can be seen as the SAT-counterpart to the partial-ordering based ILP model. We suggest a modification of the symmetry-breaking constraints used in [19] for the partial-ordering based model that can be encoded into SAT in polynomial size and without adding new variables. Furthermore, we suggest a new hybrid version inspired by the (`POPH-I`) model.

3.1 SAT encodings based on partial-ordering: (POP-S) and (POPH-S)

$$\neg y_{v,k} \qquad \forall v \in V \tag{12a}$$

$$y_{v,i} \vee \neg y_{v,i+1} \qquad \forall v \in V, i = 1, ..., k-1 \tag{12b}$$

$$y_{u,1} \vee y_{v,1} \qquad \forall \{u,v\} \in E \tag{12c}$$

$$\neg y_{u,i-1} \vee y_{u,i} \vee \neg y_{v,i-1} \vee y_{v,i} \qquad \forall \{u,v\} \in E, i = 2, ..., k \tag{12d}$$

$$y_{v,i} \in \{True, False\} \qquad \forall v \in V, i = 1, ..., k$$

The clauses (12a) guarantee that every vertex is at most as large as color k in the partial order. Clauses (12b) ensure the transitivity of the partial order, i.e., vertex v being larger than color i implies that it is also larger than color $i-1$. Finally, clauses (12c)-(12d) enforce that adjacent vertices must get a different color. In total, the model contains $k \cdot |V|$ variables and $k \cdot (|V| + |E|)$ constraints. However, one can preassign the variables according to clauses (12a), reducing the number of variables to $(k-1) \cdot |V|$ and the number of clauses to $k \cdot (|V| + |E|) - |V|$.

Note that the partial-ordering based model directly encodes that each vertex is assigned to *exactly* one color (in contrast, the assignment based model needs additional cardinality constraints to enforce this).

Adapting symmetry-breaking constraints for the POP-Model

The translation of the symmetry-breaking constraints (6a) into SAT is trivial:

$$\neg y_{v,v} \qquad \forall v \in 1, ..., k \tag{13}$$

A drawback of inequality (6b) is that translating it into a SAT encoding is no longer straightforward. However, we propose the following simplified inequality, that also eliminates all symmetries arising due to relabeling of the colors:

$$y_{v,i} \leq \sum_{u=i-1}^{v-1} y_{u,i-1} \qquad \forall v \in V \setminus \{1, |V|\}, i = 2, ..., k \tag{14}$$

▶ **Lemma 2.** *Inequalities (6a) and (14) guarantee that for all $i = 2, ..., k$, the smallest vertex in color class i is larger than the smallest vertex in color class $i-1$.*

Proof. In case $i = 2$, the claim follows directly from (6a). Assume, for contradiction, $i > 2$ is the greatest color such that for the smallest vertex v of i and the smallest vertex u of $i-1$, it holds that $v < u$. Since we have $y_{v,i-1} = 1$, according to (14) there must exist a vertex $w \in i-1, ..., v-1$, such that $y_{w,i-2} = 1$. Let w be the smallest of such vertices. From $u > v$ and from the fact that u and v are smallest vertices of colors $i-1$ and i follows that vertex w cannot be colored with $i-1$ or i. So w must be colored with a color $i^* \geq i+1$. The construction of w implies that w is the smallest one of the vertices with colors $i, i+1, ..., i^*$. It follows that w of color i^* is smaller than the smallest vertex of color $i^* - 1$. This contradicts our assumption that i is the greatest color such that the smallest vertex of i is smaller than the smallest vertex of color $i-1$. ◀

The advantage of (14) over the naive adaptation is that it can easily be encoded as a set of logical clauses:

$$\neg y_{v,i} \vee \bigvee_{u=i-1}^{v-1} y_{u,i-1} \qquad \forall v \in V \setminus \{1, |V|\}, i = 2, ..., k \tag{15}$$

The SAT encoding based on partial-ordering for the GCP has the following form:

POP-S: consists of clauses (12a)–(12d), (13) and (15).

3.1.1 Hybrid partial-ordering based SAT encoding for the GCP

One can also encode the hybrid partial-ordering based model as SAT. The clauses corresponding to (5a)-(5b) are:

$$x_{v,1} \vee y_{v,1} \qquad \forall v \in V \tag{16a}$$
$$\neg x_{v,1} \vee \neg y_{v,1} \qquad \forall v \in V \tag{16b}$$
$$\neg x_{v,i} \vee y_{v,i-1} \qquad \forall v \in V, i = 2, ..., k \tag{16c}$$
$$\neg x_{v,i} \vee \neg y_{v,i} \qquad \forall v \in V, i = 2, ..., k \tag{16d}$$
$$x_{v,i} \vee \neg y_{v,i-1} \vee y_{v,i} \qquad \forall v \in V, i = 2, ..., k \tag{16e}$$

The SAT encoding of the hybrid partial-ordering based model for the GCP has the following form:

`POPH-S`: consists of clauses (12a),(12b),(8b),(16a)-(16e), (13) and (10b).

3.2 Partial-ordering based ILP models (`POP-I-B`) and (`POPH-I-B`) for the BCP

To adapt the partial-ordering based model to the bandwidth coloring problem, one could follow the same approach that was used for the assignment model and use a constraint for every edge e and every pair of colors i, j with $|i - j| < d(e)$. However, we suggest an alternative approach, which takes advantage of the fact that the partial-ordering based model orders the vertices with respect to the colors to design a more efficient encoding. The idea of our approach is that the constraint $|c(u) - c(v)| \geq d(e)$ can equivalently be encoded as $c(u) \leq c(v) - d(e) \vee c(u) \geq c(v) + d(e)$, intuitively speaking, the color of v must be at least $d(e)$ greater or less than the color of u. This directly leads to the following implication:

$$c(u) = i \Rightarrow c(v) \leq i - d(e) \vee c(v) \geq i + d(e)$$

By definition, it holds that:

$$c(u) = i \Leftrightarrow y_{u,i-1} - y_{u,i} = 1$$
$$c(v) \leq i \Leftrightarrow y_{v,i} = 0$$
$$c(v) \geq i \Leftrightarrow y_{v,i-1} = 1$$

For the sake of convenience, we define $y_{v,i} := 1$ for $i < 1$ and $y_{v,i} := 0$ for $i > H$. Substituting the terms from the previous implication gives the following constraints:

$$y_{u,i-1} - y_{u,i} + y_{v,i-d(e)} - y_{v,i+d(e)-1} \leq 1 \qquad \forall e = \{u, v\} \in E, i = 1, ..., H. \tag{17}$$

Our new partial-ordering ILP model for the bandwidth coloring problem has the following form:

$$\texttt{POP-I-B}: \quad \min \left\{ 1 + \sum_{i=1}^{H} y_{q,i} : y \text{ satisfy (4a), (4b), (17), (4e), (4f)} \right\}.$$

▶ **Observation 3.** *By Lemma 1, the number of constraints in the assignment model (`ASS-I-B`) is* $(H + 1) \cdot |V| + H \cdot |E|(2\bar{d} - 1) - \sum_{e \in E} \big(d(e)^2 - d(e)\big) \overset{H \geq \bar{d}}{=} \mathcal{O}\big(H \cdot |E|(2\bar{d} - 1)\big)$ *and thus depends on both* $\bar{d}$ *and* H*. In contrast, the number of constraints in the partial-ordering based model (`POP-I-B`) is in the order of* $\mathcal{O}(H \cdot |E|)$ *(by straightforward counting), and thus depends only indirectly on the edge weights (via* H*). This gives a size reduction in the order of* $\mathcal{O}(\bar{d})$*. This fact applies analogously to the corresponding SAT encodings.*

3.2.1 Hybrid partial-ordering ILP model for the BCP

Analogous to the ILP models for the GCP, one can also formulate a hybrid partial-ordering based model for the BCP having less nonzero terms in the edge constraints than the regular partial-ordering based model. The edge constraints for this model are:

$$x_{ui} + y_{v,i-d(e)} - y_{v,i+d(e)-1} \leq 1 \qquad \forall e = \{u, v\} \in E, i = 1, ..., H. \tag{18}$$

The model then has the following form:

$$\texttt{POPH-I-B}: \quad \min \left\{ 1 + \sum_{i=1}^{H} y_{q,i} \colon x, y \text{ satisfy (1e), (4a), (4b), (4e)–(4f), (5a), (5b), (18)} \right\}.$$

3.3 SAT encodings (`POP-S-B`) and (`POPH-S-B`) based on partial-ordering for the BCP

The ILP formulations introduced in the previous section can easily be translated into SAT encodings. For the sake of convenience, we define $y_{v,i} := True$ for $i < 1$ and $y_{v,i} := False$ for $i > k$. The clauses corresponding to constraints (17) are:

$$\neg y_{u,i-1} \vee y_{u,i} \vee \neg y_{v,i-d(e)} \vee y_{v,i+d(e)-1} \qquad \forall e = \{u, v\} \in E, i = 1, ..., k. \tag{19}$$

which gives us the encoding `POP-S-B` as follows:

`POP-S-B`: consists of clauses (12a),(12b), and (19).

3.3.1 Hybrid partial-ordering SAT encoding for the BCP

The clauses corresponding to constraints (17) are:

$$\neg x_{u,i} \vee \neg y_{v,i-d(e)} \vee y_{v,i+d(e)-1} \qquad \forall e = \{u, v\} \in E, i = 1, ..., k. \tag{20}$$

which gives us the encoding `POPH-S-B` as follows:

`POPH-S-B`: consists of clauses (12a),(12b), (16a)–(16e) and (20).

4 Experimental evaluation

In our computational experiments, we evaluated the effectiveness of the partial-ordering based SAT encodings and compared them with state-of-the art approaches. In particular, we were interested in a comparison of the partial-ordering based encoding with the assignment based SAT encoding (i.e., the basic SAT encoding) as well as the ILP formulations of the assignment and the partial-ordering based models. Moreover, we compared the models to state-of-the-art approaches. The implementation and the data is publically available on `https://github.com/s6dafabe/popsatgcpbcp`.

4.1 Implementation details

We used the standard preprocessing techniques for graph coloring instances also used in [23, 13]:

i. A vertex u is dominated by a vertex v, $v \neq u$, if the neighborhood of u is a subset of the neighborhood of v. In this case, the vertex u can be deleted from G, the remaining graph can be colored, and at the end, u can get the same color as v.

ii. If a vertex v has a degree of less than L, where L is a lower bound on the chromatic number, then v can be deleted from G for the calculations. At the end, after the remaining graph has been colored, there is at least one used color left to color v that is not assigned to any of the neighbors of v.

iii. Any clique Q represents a lower bound, so one can precolor the vertices in a clique with colors $1, ..., |Q|$, eliminating some of the variables. To fix as many variables as possible, one tries to find a clique Q of maximum size.

To reduce the graph as much as possible, we use reductions (i) and (ii) alternatingly until the graph cannot be reduced further. To compute the clique for (iii), we apply the randomised function `networkx.maximal_independent_set()` on the complement graph of G and choose the best clique out of $300 \cdot \frac{|E|}{|V|}$ iterations. Another refinement we use is that of all the largest cliques found, we use the one that has the largest cut. The motivation behind this is that precoloring a vertex v with a color i also fixes some variables of their neighbors, as it excludes coloring the neighbors of v with i. As the clique finding method is time consuming for large graphs, we limit the time for finding a clique to $100s$, after which we use the best clique found so far.

Because SAT is a decision problem, we need to solve a series of k-colorability problems to find the chromatic number of a graph. We found that using ascending linear search, i.e. starting from a lower bound $L(G)$ and testing satisfiability for $k = L(G), L(G) + 1, ..., \chi(G)$ until the first satisfiable value for k is found, leads to the best results for the graph coloring problem. For the lower bound $L(G)$, we use the size of the clique found in preprocessing step (3). For the bandwidth coloring problem, we found that descending linear search, i.e. testing $k = H(G), H(G) - 1, ..., \chi(G)$ leads to the best result. To compute an upper bound $H(G)$ for the optimal value for the BCP, we use a simple greedy algorithm: In every iteration we select the vertex that has not yet been assigned a color and has the highest degree. We then assign the vertex to the smallest color that does not conflict with the colors of any neighbouring vertices that have already been colored.

Note that we omitted the preprocessing steps (i)-(iii) for the BCP, as they are not applicable for this problem: Because of the distance constraints, swapping the indices between colors may invalidate a coloring, therefore these colorings are not equivalent anymore. Similarly, fixing the colors of vertices in a clique may lead to the optimal solution being excluded.

4.2 Test setup and benchmark set

To solve the SAT encodings, we used the solver `kissat 3.1.1`[1], which has been successful in the 2022 SAT competition. The preprocessing and the generation of the SAT and ILP formulations were implemented in `Python 3.10` using the library `networkx 2.8.5`[2]. For solving the ILP models, we used `Gurobi 10.0.2` single-threadedly. The machine used to evaluate the SAT and ILP formulations features an Intel Xeon Gold 6130@2.1GHz running CentOS Linux and 187 GB of memory (Benchmarks [2] user time: r500.5=4.87s). For comparison, we compiled the implementation of Heule, Karahalios and Hoeve [11] and tested it on the same machine. We also compiled the implementation of Held, Cook and Sewell [10] with the same Gurobi version. Because their implementation[3] uses features of Gurobi that are

[1] `https://github.com/arminbiere/kissat/releases/tag/rel-3.1.1`
[2] `https://networkx.org/documentation/stable/release/release_2.8.5.html`
[3] `https://github.com/heldstephan/exactcolors`

incompatible with the first machine, this method had to be evaluated on a different machine having an AMD EPYC 7543P@2.8GHz and 257GB memory (Benchmarks [2] user time: r500.5=3.24s). For the graph coloring problem, we performed our experiments on a set of 134 DIMACS graphs [22] and additionally a set of 9 randomly generated instances by Michael Trick (the `R`-instances: Note that there exist 18 instances in total, however the instances are duplicated and the duplicates only differ in the node weights, which are irrelevant for standard graph coloring). Furthermore, we compare with the results reported in [11] of the method presented in [8]. For similar reasons as reported in Heule, Karahalios and van Hoeve [11], we did not compare to the work in [7], which claims strong results for graph coloring with a method using a relaxed Zykov encoding that is incrementally strengthened. The linked source code is currently incorrect, and the authors were unable to reproduce the results. For the bandwidth coloring problem, we used the `GEOM` set consisting of 33 graphs generated by Michael Trick [22]. We have set a time limit of 1 hour.

4.3 Experimental results for the graph coloring problem

Table 1 Number of solved DIMACS instances on the benchmark set for the GCP.

set	`ASS-S`	`POP-S`	`POPH-S`	`ASS-I`	`POP-I`	`POPH-I`	EC[10]	CLICOL [11]	CDCL[8, 11] [4]
DSJ	2	2	2	3	2	3	5	4	2
FullIns	14	14	14	12	11	12	5	14	14
Insertions	4	4	4	4	4	4	1	4	3
abb313GPI	1	1	1	0	0	0	0	0	0
anna	1	1	1	1	1	1	1	1	1
ash	3	3	3	3	3	3	0	3	3
david	1	1	1	1	1	1	1	1	1
flat	0	0	0	0	0	0	2	0	0
fpsol2	3	3	3	3	1	3	3	3	3
games120	1	1	1	1	1	1	1	1	1
homer	1	1	1	1	1	1	1	1	1
huck	1	1	1	1	1	1	1	1	1
inithx	3	3	3	3	1	3	3	3	3
jean	1	1	1	1	1	1	1	1	1
latin_square	0	0	0	0	0	0	0	0	0
le450	8	8	8	8	7	8	3	10	8
miles	5	5	5	5	5	5	5	5	5
mug	4	4	4	4	4	4	4	4	4
mulsol	5	5	5	5	4	5	5	5	5
myciel	4	4	4	3	3	3	2	4	4
r	8	7	7	7	5	7	7	7	7
qg	3	3	3	1	0	2	0	3	3
queen	6	**8**	**8**	6	5	6	7	5	6
school1	2	2	2	2	2	2	1	2	1
wap0	3	**5**	**5**	0	0	1	1	4	1
will199GPI	1	1	1	1	1	1	1	1	1
zeroin	3	3	3	3	3	3	3	3	3
total	88	**91**	**91**	79	67	81	64	90	83

Table 2 Number of solved instances on the R-instances.

set	ASS-S	POP-S	POPH-S	ASS-I	POP-I	POPH-I	EC[10]	CLICOL [11]
R50	3	3	3	3	3	3	3	3
R75	2	2	2	2	1	2	3	3
R100	2	2	2	2	1	2	3	2
total	7	7	7	7	5	7	9	8

Table 1 shows the number of solved instances for the 134 evaluated DIMACS instances for the SAT encodings of the assignment (`ASS-S`), the partial-ordering (`POP-S`), and the hybrid partial-ordering (`POPH-S`) based models, the corresponding ILP formulations (`ASS-I`), (`POP-I`), (`POPH-I`), the method by Held, Cook, and Sewel [10] (`EC`), the method of Heule, Karahalios and van Hoeve [11] (`CLICOL`) and the results of the method of Hebrard and Katsirelos [8] (`CDCL`) as reported in [11]. The first column of the table describes the class type and the subsequent columns show the number of solved instances for each model, out of the total number of tested instances. Table 2 shows the number of solved instances for the 9 `R`-instances. Unfortunatly, the code provided [8] did not compile on our machine, therefore we used the results of the experiments performed in [11] for the algorithm `CDCL`. Note that the authors in [11] did not evaluate on the `R` instances, which is why they are missing in table 2. Note that for the ILP models, the large instances `DSJC1000.9`, `latin_square_10` and `qg.order100` resulted in out-of-memory exceptions. Also, the algorithm `CLICOL` does not seem to be robust for large instances, as it produced runtime errors for the instances `r1000.5`, `latin_square_10` and `wap04a`. The bold items in the table highlight the instance types for which the POP encodings solved more instances than the other methods.

We can see that (`POP-S`), (`POPH-S`) solved the most DIMACS instances (91/134), followed closely by `CLICOL` (90/134). Furthermore, the POP based SAT encodings were able to solve three more instances than (`ASS-S`), and there was only one instance that was solved by (`ASS-S`) but not by the POP based SAT encodings. The POP encodings performed especially well on the `wap0`-class, as they solved the instances `wap01a`, `wap02a` and `wap08a`, which were only closed recently [11].

For the `R`-instances, we can see that all four SAT methods perform similarly, with `CLICOL` slightly outperforming the three simple encodings. (`ASS-I`) and (`POPH-I`) also solve the same amount of instances as the three evaluated SAT encodings. `EC` performs the best, solving all instances, while (`POP-I`) performs the worst, which is to be expected, as the instance set contains many instances with medium and high density, for which `EC` typically performs well and (`POP-I`) typically performs poorly.

It can be observed that the SAT encodings generally outperform the ILP formulations: All of the three evaluated SAT encodings solve more instances than all three ILP formulations, even though the underlying models are very similiar. A reason for this observed behaviour is that the LP-relaxations of the assignment and the partial-ordering based models are very weak, which in turn causes the lower bounds derived from the LP-relaxations to be weak. ILP solvers spend a lot of time calculating LP-relaxations during branching to bound the search tree, however, as argued before this technique is not effective for these particular formulations. On the other hand, the clause-learning methods employed by modern SAT-Solvers may work better in this context because they do not rely on the strength of the LP-Relaxation.

[4] The code provided in the repository produced compile errors on our system, so we used the results of the experiments from [11] which did not contain the `R`-instances.

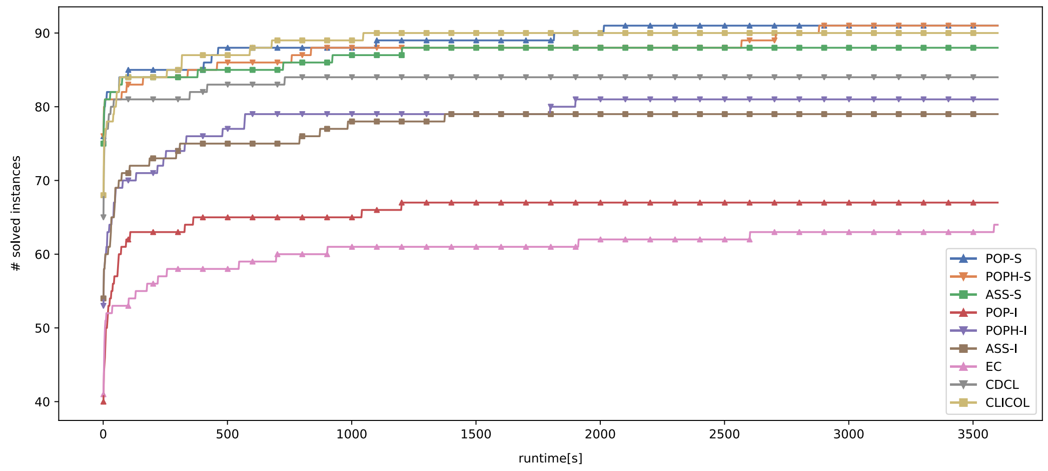


Figure 1 Number of DIMACS instances solved within a given runtime for the GCP.

Figure 1 visualizes for each model the number of instances, which can be solved within a time limit of 1, 2,..., 3600 seconds. We omitted the `R` instances in this figure for better comparability. We can see that (`POP-S`) and (`POPH-S`) solve more instances than (`ASS-S`) independent of the considered time limit. Generally, (`POP-S`), (`POPH-S`) and `CLICOL` are the best approaches and perform similarly. An interesting observation is that for the ILP formulations, the POP formulation performs far worse than the other two formulations (`ASS-I`) and (`POPH-I`), while for the SAT encodings, (`POP-S`) and (`POPH-S`) show almost identical performance (with (`POP-S`) even being slightly better). Jabrayilov and Mutzel [13] argued that one weakness of the POP ILP formulaton lies in the denser constraint matrix, which is caused by the POP model containing 4 variables in the constraints enforcing differing colors for connected vertices. However, this does not seem to impact the performance of the SAT encoding.

In total (for DIMACS and `R`-instances combined), (`POP-S`), (`POPH-S`) and `CLICOL` solved the most instances (98/143). One interesting thing to note is that although (`POP-S`)/(`POPH-S`) and `CLICOL` solved the same number of instances, they solved a different set of instances. For example (`POP-S`)/(`POPH-S`) is particularly advantageous on the `queen` instances, while `CLICOL` shows superior performance on the `le450` instances. We want to remark that the `CLICOL` approach uses the assignment-based SAT encoding as a sub-algorithm and combines it with a more sophisticated method of finding an initial clique used for precoloring. An interesting idea could be to use the partial-ordering based SAT encoding in the `CLICOL` framework to try and combine the advantages of both methods.

4.4 Experimental results for the bandwidth coloring problem

Table 3 shows the number of solved instances for the 33 evaluated bandwidth coloring instances for the SAT encodings of the assignment (`ASS-S-B`), the partial-ordering (`POP-S-B`), the hybrid partial-ordering (`POPH-S-B`) based models, the corresponding ILP formulations (`ASS-I-B`), (`POP-I-B`), (`POPH-I-B`), and the constraint programming results of the method by Dias et al. [5] from the literature. Note that the ILP formulation of the assignment model is equivalent to the model MinGEQ-CDGP-IP used in [5].

[5] The authors did not provide the code, so we used the results as reported in [5]. We remark that the used time limit in the paper was 24 hour compared to 1 hour in our experiments.

Table 3 Number of solved GEOM instances for the BCP.

set	#inst.	ASS-S-B	POP-S-B	POPH-S-B	ASS-I-B	POP-I-B	POPH-I-B	DFMM[5] [5]
GEOM[20-50]	4	4	4	4	4	4	4	4
GEOM[20a-50a]	4	4	4	4	4	4	3	4
GEOM[20b-50b]	4	4	4	4	4	4	4	4
GEOM[60-90]	4	4	4	4	4	4	4	4
GEOM[60a-90a]	4	4	4	4	1	1	0	4
GEOM[60b-90b]	4	3	**4**	**4**	0	0	1	3
GEOM[100-120]	3	3	3	3	3	0	0	1
GEOM[100a-120a]	3	0	**3**	**3**	0	0	0	1
GEOM[100b-120b]	3	0	**2**	**2**	0	0	0	1
#solved	33	26	**32**	**32**	20	17	16	26

We can see that (POP-S-B) and (POPH-S-B) solve the most instances, followed by (ASS-S-B) and the constraint programming approach used in [5]. Note that the time limit used in [5] is 24 hours compared to just 1 hour in our experiments. Interestingly, one can observe an opposite trend for the ILP formulations, where the POP formulations are weaker than the assignment formulations. This may be caused by the denser constraint matrices of the POP formulations as argued before, which do not have an impact on the performance of the SAT encodings.

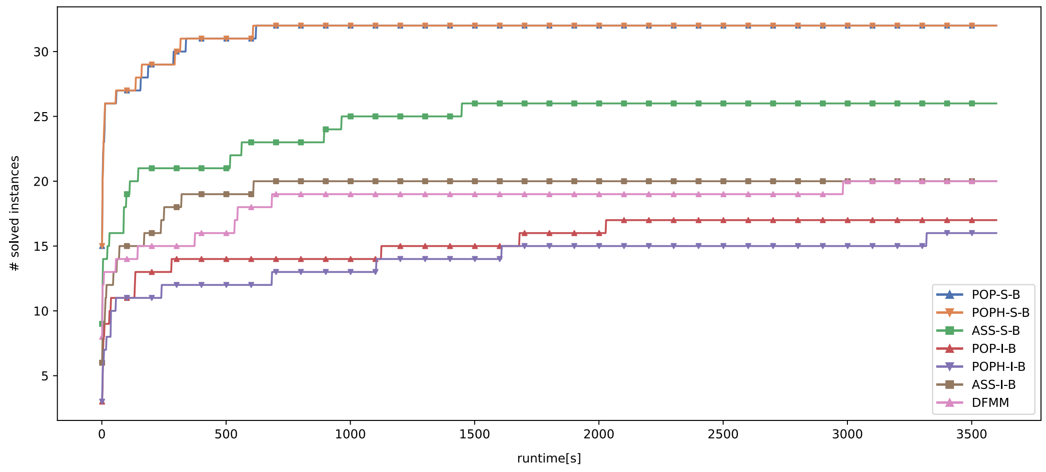


Figure 2 Number of GEOM instances solved within a given runtime for the BCP.

Figure 2 shows the number of solved instances within a time limit of 1, 2,..., 3600 seconds. One can see that the performance of (POP-S-B) and (POPH-S-B) is nearly identical and that the two models dominate the other approaches. In particular, the second best approaches (ASS-S-B and constraint programming) solved 26 instances in total, which (POP-S-B) and (POPH-S-B) both solved in less than 10 seconds; after less than 700 seconds, the POP encodings solved all but one of the 33 GEOM instances to optimality. To our knowledge, this is the first time the instances GEOM90b, GEOM100a, GEOM100b, GEOM110a, GEOM110b and GEOM120a were solved to optimality.

5 Conclusion

In this paper, we have revisited the partial-ordering based ILP and SAT formulations for the graph coloring problem and have suggested new models for the bandwidth coloring problem based on partial-ordering models.

Our computational study on the graph coloring problem suggests that all three SAT encodings perform similar, with (`POP-S`) and (`POPH-S`) solving slightly more instances (98/143) than (`ASS-S`) (95/143). This holds true for every timelimit up to 1 hour. Moreover, the SAT encodings solve more instances than the ILP formulations. Compared to state-of-the-art approaches, the tested SAT based approaches solved more instances than the approach based on the set cover ILP formulations and have shown to be particularly advantageous for sparse graphs. Moreover, the tested SAT based encodings also solved more instances than reported in [8] and the same amount of instances as [11]. Specifically, (`POP-S`) and (`POPH-S`) have proven to be effective on the `wap0`- and `queen`-instances. We also remark that the partial-ordering based encodings are as easy to use as the classical assignment-based encodings. As (`POP-S`) and (`POPH-S`) have shown superior performance compared to (`ASS-S`), an interesting line of research could therefore be to incorporate the encodings into other SAT-based frameworks, such as the method presented in [11].

Concerning the bandwidth coloring problem, we have seen that the new POP-based SAT formulations dominate the exact state-of-the-art methods. Compared to the ILP formulations and the constraint programming approach, the SAT-encodings of the POP-based model solve the most instances by far and have a significantly lower runtime on a large part of the instances. This is consistent with the theoretical advantage of the partial-ordering based model, which has an asymptotically smaller formulation size compared to the assignment based model.

References

1. Carlos Ansótegui and Felip Manyà. Mapping problems with finite-domain variables into problems with boolean variables. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL: `http://www.satisfiability.org/SAT04/programme/53.pdf`.
2. Benchmarking machines and testing solutions. `http://mat.gsia.cmu.edu/COLOR02/BENCHMARK/benchmark.tar`, 2002.
3. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981. `doi:10.1016/0096-0551(81)90048-5`.
4. Munmun Dey and Amitava Bagchi. Satisfiability methods for colouring graphs. In *ACER 2013*, volume 3, pages 135–147, March 2013. `doi:10.5121/csit.2013.3213`.
5. Bruno Dias, Rosiane De Freitas Rodrigues, Nelson Maculan, and Philippe Michelon. Integer and constraint programming approaches for providing optimality to the bandwidth multicoloring problem. *RAIRO - Operations Research*, 55:S1949–S1967, June 2021. `doi:10.1051/ro/2020065`.
6. Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? Graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005. `doi:10.1137/S0036144504444711`.
7. Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Nicolas Szczepanski. An incremental SAT-based approach to the graph colouring problem. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 213–231. Springer, 2019. `doi:10.1007/978-3-030-30048-7_13`.

8 Emmanuel Hebrard and George Katsirelos. Clause learning and new bounds for graph coloring. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 6166–6170, July 2019. doi:10.24963/ijcai.2019/856.

9 Emmanuel Hebrard and George Katsirelos. Constraint and satisfiability reasoning for graph coloring. *J. Artif. Intell. Res.*, 69:33–65, 2020. doi:10.1613/JAIR.1.11313.

10 Stephan Held, William Cook, and Edward Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4, December 2012. doi:10.1007/s12532-012-0042-3.

11 Marijn J. H. Heule, Anthony Karahalios, and Willem-Jan van Hoeve. From Cliques to Colorings and Back Again. In Christine Solnon, editor, *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:10, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CP.2022.26.

12 Thore Husfeldt. Graph colouring algorithms, 2015. arXiv:1505.05825.

13 Adalat Jabrayilov and Petra Mutzel. New integer linear programming models for the vertex coloring problem. In Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro, editors, *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings*, volume 10807 of *Lecture Notes in Computer Science*, pages 640–652. Springer, 2018. doi:10.1007/978-3-319-77404-6_47.

14 Adalat Jabrayilov and Petra Mutzel. Strengthened partial-ordering based ILP models for the vertex coloring problem. *CoRR*, abs/2206.13678, 2022. doi:10.48550/arXiv.2206.13678.

15 Frank Thomson Leighton. A graph coloring algorithm for large scheduling problems. *Journal of research of the National Bureau of Standards*, 84 6:489–506, 1979. URL: https://api.semanticscholar.org/CorpusID:16043293.

16 Enrico Malaguti and Paolo Toth. A survey on vertex coloring problems. *Int. Trans. Oper. Res.*, 17:1–34, 2010. URL: https://api.semanticscholar.org/CorpusID:9199689.

17 Rafael Marti, Francisco Gortázar, and Abraham Duarte. Heuristics for the bandwidth colouring problem. *Int. J. Metaheuristics*, 1:11–29, January 2010. doi:10.1504/IJMHEUR.2010.033121.

18 Anuj Mehrotra and Michael A. Trick. A column generation approach for graph coloring. *INFORMS J. Comput.*, 8:344–354, 1996. URL: https://api.semanticscholar.org/CorpusID:13978746.

19 Isabel Méndez-Díaz and Paula Zabala. A cutting plane algorithm for graph coloring. *Discrete Applied Mathematics*, 156(2):159–179, 2008. Computational Methods for Graph Coloring and it's Generalizations. doi:10.1016/j.dam.2006.07.010.

20 Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005. doi:10.1007/11564751_73.

21 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.

22 Michael Trick. DIMACS graph coloring instances. https://mat.tepper.cmu.edu/COLOR04/, 2002.

23 Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008. Computational Methods for Graph Coloring and it's Generalizations. doi:10.1016/j.dam.2006.07.016.

A Detailed results of the nine models for the 143 DIMACS instances for the GCP

Table 4 Results of the nine models for the 134 DIMACS instances and the 9 R-instances for the GCP.

			POP-S			POPH-S			ASS-S			POP-I			POPH-I			ASS-I			EC	CLICOL	CDCL [6]
Instance	V	E	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	time[s]	time[s]	time[s]
1-FullIns_3	30	100	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	0.0	0.2	0
1-FullIns_4	93	593	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.1	5	5	0.2	5	5	0.1	3600.1	0.1	0
1-FullIns_5	282	3247	6	6	0.5	6	6	0.6	6	6	0.5	6	6	361.2	6	6	242.9	6	6	29.4	3600.5	0.3	0
1-Insertions_4	67	232	5	5	1.4	5	5	1.2	5	5	1.0	5	5	71.7	5	5	480.4	5	5	106.9	3600.1	1.9	3600.0
1-Insertions_5	202	1227	4	6	3600.0	4	6	3600.0	4	6	3600.0	4	6	3600.0	4	6	3600.0	4	6	3600.0	3602.6	3600.0	3600.0
1-Insertions_6	607	6337	4	7	3600.0	4	7	3600.0	4	7	3600.0	4	7	3600.1	4	7	3600.0	4	7	3600.0	3600.0	3600.0	3600.0
2-FullIns_3	52	201	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	0.0	0.5	0
2-FullIns_4	212	1621	6	6	0.0	6	6	0.0	6	6	0.1	6	6	1.8	6	6	0.4	6	6	0.2	3600.1	0.1	0
2-FullIns_5	852	12201	7	7	0.7	7	7	0.6	7	7	0.8	6	7	3600.0	7	7	1899.5	7	7	983.6	3602.5	3.2	20
2-Insertions_3	37	72	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.1	4	4	0.1	4	4	0.1	220.2	0.1	0
2-Insertions_4	149	541	3	5	3600.0	3	5	3600.0	3	5	3600.0	4	5	3600.0	4	5	3600.0	4	5	3600.0	3601.5	3600.0	3600.0
2-Insertions_5	597	3936	3	6	3600.0	3	6	3600.0	3	6	3600.0	3	6	3600.1	3	6	3600.0	3	6	3600.0	3600.0	3600.0	3600.0
3-FullIns_3	80	346	6	6	0.1	6	6	0.1	6	6	0.1	6	6	0.0	6	6	0.0	6	6	0.0	0.0	0.5	0
3-FullIns_4	405	3524	7	7	0.1	7	7	0.1	7	7	0.1	7	7	7.6	7	7	1.4	7	7	0.4	3600.2	0.4	0
3-FullIns_5	2030	33751	8	8	4.1	8	8	5.9	8	8	5.8	7	8	3600.0	7	8	3600.0	7	8	3600.0	3602.9	43.0	0
3-Insertions_3	56	110	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.8	4	4	1.1	4	4	1.4	3600.1	0.2	1
3-Insertions_4	281	1046	3	5	3600.0	3	5	3600.0	3	5	3600.0	3	5	3600.0	4	5	3600.0	4	5	3600.0	3613.8	3600.0	3600.0
3-Insertions_5	1406	9695	3	6	3600.0	3	6	3600.0	3	6	3600.0	3	6	3600.1	3	6	3600.0	3	6	3600.0	3600.0	3600.0	3600.0
4-FullIns_3	114	541	7	7	0.1	7	7	0.1	7	7	0.0	7	7	0.0	7	7	0.0	7	7	0.0	0.0	0.2	0
4-FullIns_4	690	6650	8	8	0.4	8	8	0.5	8	8	0.6	8	8	60.1	8	8	0.6	8	8	0.5	3600.7	1.2	0
4-FullIns_5	4146	77305	9	9	63.8	9	9	93.3	9	9	75.6	8	9	3600.0	8	9	3600.0	8	9	3600.0	3629.8	678.1	730
4-Insertions_3	79	156	4	4	0.1	4	4	0.1	4	4	0.2	4	4	17.5	4	4	15.7	4	4	48.3	3600.2	0.2	417
4-Insertions_4	475	1795	3	5	3600.0	3	5	3600.0	3	5	3600.0	3	5	3600.0	3	5	3600.0	3	5	3600.0	3600.0	3600.0	3600.0
5-FullIns_3	154	792	8	8	0.2	8	8	0.1	8	8	0.2	8	8	0.0	8	8	0.0	8	8	0.0	0.0	0.4	0
5-FullIns_4	1085	11395	9	9	1.1	9	9	1.4	9	9	3.1	9	9	31.8	9	9	0.9	9	9	0.9	3600.3	7.0	0
abb313GPIA	1555	53356	9	9	0.5	9	9	0.2	9	9	0.2	8	9	3600.1	8	10	3600.0	8	9	3600.0	3600.0	3600.0	3600.0
anna	138	493	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	0.0	0.4	0
ash331GPIA	662	4181	4	4	0.1	4	4	0.1	4	4	0.1	4	4	6.3	4	4	16.0	4	4	28.6	3611.7	0.2	0
ash608GPIA	1216	7844	4	4	0.1	4	4	0.1	4	4	0.0	4	4	10.7	4	4	34.1	4	4	74.3	3600.0	0.5	4
ash958GPIA	1916	12506	4	4	0.1	4	4	0.1	4	4	0.1	4	4	57.5	4	4	217.9	4	4	872.6	3600.0	1.4	29
david	87	406	11	11	0.1	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.0	0.0	0.0	0
DSJC1000.1	1000	49629	6	27	3600.0	6	27	3600.0	6	27	3600.0	6	0	3600.0	6	0	3600.0	6	0	3600.0	3600.0	3600.0	3600.0
DSJC1000.5	1000	249826	16	115	3600.0	16	115	3600.0	16	115	3600.0	14	0	3603.0	14	0	3600.1	14	0	3600.8	3600.0	3600.0	3600.0
DSJC1000.9	1000	449449	60	299	3600.0	59	299	3600.0	59	299	3600.0	-	-	-	-	-	-	-	-	-	3600.0	3600.0	3600.0
DSJC125.1	125	736	5	5	0.0	5	5	0.1	5	5	0.1	5	5	1.4	5	5	2.5	5	5	2.1	102.7	0.1	0
DSJC125.5	125	3891	13	22	3600.0	13	22	3600.0	13	22	3600.0	11	20	3600.0	13	0	3600.0	13	19	3600.0	3600.5	3600.0	3600.0
DSJC125.9	125	6961	38	51	3600.0	38	51	3600.0	38	51	3600.0	35	47	3600.0	41	44	3600.0	42	44	3600.0	6.2	3600.0	3600.0
DSJC250.1	250	3218	5	10	3600.0	6	10	3600.0	6	10	3600.0	5	10	3600.0	5	10	3600.0	5	10	3600.0	3600.0	3600.0	3600.0
DSJC250.5	250	15668	14	37	3600.0	14	37	3600.0	14	37	3600.0	12	0	3600.0	15	0	3600.0	14	0	3600.0	3606.8	3600.0	3600.0
DSJC250.9	250	27897	44	92	3600.0	44	92	3600.0	44	92	3600.0	41	0	3600.1	47	0	3600.0	43	0	3600.0	3604.0	3600.0	3600.0
DSJC500.1	500	12458	6	16	3600.0	6	16	3600.0	6	16	3600.0	6	0	3600.0	6	0	3600.0	6	0	3600.0	3600.0	3600.0	3600.0
DSJC500.5	500	62624	15	65	3600.0	15	65	3600.0	15	65	3600.0	13	0	3600.2	14	0	3600.0	13	0	3600.0	3600.0	3600.0	3600.0
DSJC500.9	500	112437	53	170	3600.0	53	170	3600.0	53	170	3600.0	48	0	3600.1	47	0	3600.0	47	0	3600.0	3637.7	3600.0	3600.0
DSJR500.1	500	3555	12	12	0.0	12	12	0.0	12	12	0.0	12	12	1.2	12	12	0.3	12	12	0.2	255.1	0.9	0
DSJR500.1c	500	121275	81	89	3600.0	80	89	3600.0	80	89	3600.0	74	0	3600.1	81	0	3600.0	75	89	3600.3	695.1	52.8	3600.0
DSJR500.5	500	58862	122	131	3600.0	122	131	3600.0	122	131	3600.0	115	0	3600.1	122	122	334.5	122	122	1373.1	2602.2	65.8	3600.0
flat1000_50_0	1000	245000	16	114	3600.0	16	114	3600.0	16	114	3600.0	14	0	3601.3	14	0	3600.0	14	0	3600.7	3600.0	3600.0	3600.0
flat1000_60_0	1000	245830	16	114	3600.0	16	114	3600.0	16	114	3600.0	13	0	3601.2	13	0	3600.0	13	0	3600.0	3600.0	3600.0	3600.0
flat1000_76_0	1000	246708	16	115	3600.0	16	115	3600.0	16	115	3600.0	13	0	3601.1	13	0	3600.3	13	0	3600.8	3600.0	3600.0	3600.0
flat300_20_0	300	21375	14	42	3600.0	14	42	3600.0	14	42	3600.0	11	0	3600.2	13	0	3600.0	13	0	3600.0	545.8	3600.0	3600.0
flat300_26_0	300	21633	14	41	3600.0	14	41	3600.0	14	41	3600.0	12	0	3600.1	13	0	3600.0	13	0	3600.0	901.4	3600.0	3600.0
flat300_28_0	300	21695	14	42	3600.0	14	42	3600.0	14	42	3600.0	12	0	3600.1	14	0	3600.0	14	0	3600.0	3613.3	3600.0	3600.0
fpsol2.i.1	269	11654	65	65	0.1	65	65	0.1	65	65	0.1	65	65	1.2	65	65	0.3	65	65	0.3	0.4	0.6	0
fpsol2.i.2	363	8691	30	30	0.1	30	30	0.2	30	30	0.0	28	30	3600.0	30	30	0.2	30	30	0.4	0.4	0.3	0
fpsol2.i.3	363	8688	30	30	0.0	30	30	0.0	30	30	0.0	29	30	3600.0	30	30	0.2	30	30	0.4	0.4	0.1	0
games120	120	638	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	9	9	0.0	0.0	0.1	0
homer	556	1629	13	13	0.0	13	13	0.0	13	13	0.0	13	13	61.2	13	13	0.1	13	13	0.1	0.1	0.30	0
huck	74	301	11	11	0.0	11	11	0.0	11	11	0.0	11	11	0.2	11	11	0.0	11	11	0.0	0.0	0.1	0
inithx.i.1	519	18707	54	54	0.1	54	54	0.0	54	54	0.1	54	54	0.8	54	54	0.3	54	54	0.3	1.3	0.3	0
inithx.i.2	558	13979	31	31	0.1	31	31	0.0	31	31	0.1	29	31	3600.0	31	31	0.2	31	31	0.4	0.2	0.4	0
inithx.i.3	559	13969	31	31	0.1	31	31	0.2	31	31	0.1	29	31	3600.0	31	31	0.2	31	31	0.3	0.3	0.2	0
jean	77	254	10	10	0.0	10	10	0.1	10	10	0.0	10	10	0.0	10	10	0.0	10	10	0.0	0.0	0.1	0
latin_square_10	900	307350	90	132	3600.0	90	132	3600.0	90	132	3600.0	-	-	-	-	-	-	-	-	-	3638.4	-	3600.0
le450_15a	450	8168	15	15	0.8	15	15	0.3	15	15	0.4	15	16	3600.1	15	15	328.1	15	15	307.6	3633.7	1.1	4
le450_15b	450	8169	15	15	0.7	15	15	0.2	15	15	0.2	15	15	1039.7	15	15	567.0	15	15	293.2	3600.0	0.3	1
le450_15c	450	16680	15	23	3600.0	15	23	3600.0	15	23	3600.0	15	23	3600.0	15	0	3600.0	15	0	3600.7	3600.0	63.6	3600.0
le450_15d	450	16750	15	24	3600.0	15	24	3600.0	15	24	3600.0	15	0	3600.0	15	0	3600.0	15	0	3600.4	3600.0	53.8	3600.0
le450_25a	450	8260	25	25	0.1	25	25	0.1	25	25	0.1	25	25	26.8	25	25	1.3	25	25	0.6	1.8	0.2	0
le450_25b	450	8263	25	25	0.0	25	25	0.0	25	25	0.0	25	25	10.9	25	25	0.8	25	25	0.6	1.9	0.2	0

[6] The code provided in the repository produced compile errors on our system, so we used the results of the experiments from [11] which did not contain the R-instances.

Table 4 (continued)

			POP-S			POPH-S			ASS-S			POP-I			POPH-I			ASS-I			EC	CLICOL	CDCL
Instance	V	E	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	time[s]	time[s]	time[s]
le450_25c	450	17343	25	29	3600.0	25	29	3600.0	25	29	3600.0	25	29	3600.0	25	28	3600.0	25	29	3600.1	3690.4	3600.0	3600.0
le450_25d	450	17425	25	29	3600.0	25	29	3600.0	25	29	3600.0	25	29	3600.1	25	28	3600.0	25	29	3600.0	3637.8	3600.0	3600.0
le450_5a	450	5714	5	5	0.1	5	5	0.0	5	5	0.0	5	5	108.5	5	5	23.2	5	5	62.9	3600.0	0.1	41
le450_5b	450	5734	5	5	0.0	5	5	0.0	5	5	0.0	5	5	328.8	5	5	41.2	5	5	47.6	3600.0	0.1	9
le450_5c	450	9803	5	5	0.0	5	5	0.1	5	5	0.0	5	5	92.4	5	5	47.5	5	5	19.3	3600.0	0.1	2
le450_5d	450	9757	5	5	0.1	5	5	0.1	5	5	0.0	5	5	19.8	5	5	49.8	5	5	46.8	3584.7	0.1	2
miles1000	128	3216	42	42	0.1	42	42	0.1	42	42	0.1	42	42	0.5	42	42	0.1	42	42	0.1	0.1	0.6	0
miles1500	128	5198	73	73	0.1	73	73	0.2	73	73	0.2	73	73	0.5	73	73	0.3	73	73	0.3	0.1	0.1	0
miles250	125	387	8	8	0.1	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	0.0	0.1	0
miles500	128	1170	20	20	0.1	20	20	0.0	20	20	0.1	20	20	0.0	20	20	0.0	20	20	0.0	0.0	0.1	0
miles750	128	2113	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.1	31	31	0.1	31	31	0.1	0.0	0.1	0
mug100_1	100	166	4	4	0.1	4	4	0.0	4	4	0.0	4	4	0.2	4	4	0.1	4	4	0.2	0.6	0.1	0
mug100_25	100	166	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.2	4	4	0.4	4	4	0.2	0.5	0.1	0
mug88_1	88	146	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.1	4	4	0.1	4	4	0.2	0.3	0.1	0
mug88_25	88	146	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.2	4	4	0.1	4	4	0.2	0.3	0.1	0
mulsol.i.1	138	3925	49	49	0.0	49	49	0.1	49	49	0.1	49	49	0.4	49	49	0.1	49	49	0.1	0.1	0.1	0
mulsol.i.2	173	3885	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.3	31	31	0.1	31	31	0.1	0.0	0.1	0
mulsol.i.3	174	3916	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.2	31	31	0.1	31	31	0.1	0.0	0.2	0
mulsol.i.4	175	3946	31	31	0.0	31	31	0.0	31	31	0.0	31	31	0.4	31	31	0.1	31	31	0.1	0.0	0.1	0
mulsol.i.5	176	3973	31	31	0.0	31	31	0.0	31	31	0.0	30	31	3600.0	31	31	0.1	31	31	0.1	0.0	0.1	0
myciel3	11	20	4	4	0.0	4	4	0.1	4	4	0.0	4	4	0.0	4	4	0.0	4	4	0.0	0.0	0.1	0
myciel4	23	71	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.1	5	5	0.1	5	5	0.1	4.8	0.1	0
myciel5	47	236	6	6	0.2	6	6	0.2	6	6	0.2	6	6	37.5	6	6	42.0	6	6	44.3	3600.0	0.5	0
myciel6	95	755	7	7	99.4	7	7	74.3	7	7	63.6	6	7	3600.0	6	7	3600.0	5	7	3600.0	3600.1	1045.3	0
myciel7	191	2360	6	8	3600.0	6	8	3600.0	6	8	3600.0	5	8	3600.0	5	8	3600.0	5	8	3600.0	3600.7	3600.0	0
qg.order100	10000	990000	100	116	3600.0	100	116	3600.0	100	116	3600.0	-	-	-	-	-	-	-	-	-	3600.0	3600.0	3600.0
qg.order30	900	26100	30	30	0.8	30	30	0.4	30	30	0.5	30	35	3600.2	30	30	77.6	30	30	186.5	3600.0	4.5	0
qg.order40	1600	62400	40	40	7.5	40	40	1.9	40	40	2.4	40	0	3600.0	40	40	1798.7	40	43	3600.2	3600.0	256.4	8
qg.order60	3600	212400	60	60	1813.9	60	60	835.9	60	60	28.8	60	0	3600.1	60	62	3601.5	60	0	3600.0	3600.0	315.9	347
queen10_10	100	1470	11	11	436.4	11	11	758.5	10	14	3600.0	10	12	3600.0	10	11	3600.0	10	12	3600.0	130.3	3600.0	3600.0
queen11_11	121	1980	11	11	404.3	11	11	2705.2	11	15	3600.0	11	13	3600.0	11	13	3600.0	11	13	3600.0	3602.3	3600.0	3600.0
queen12_12	144	2596	12	16	3600.0	12	16	3600.0	12	16	3600.0	12	15	3600.0	12	14	3600.0	12	14	3600.0	3604.7	3600.0	3600.0
queen13_13	169	3328	13	17	3600.0	13	17	3600.0	13	17	3600.0	13	16	3600.0	13	16	3600.0	13	15	3600.0	3602.3	3600.0	3600.0
queen14_14	196	4186	14	19	3600.0	14	19	3600.0	14	19	3600.0	14	17	3600.0	14	17	3600.0	14	17	3600.0	3602.4	3600.0	3600.0
queen15_15	225	5180	15	21	3600.0	15	21	3600.0	15	21	3600.0	15	19	3600.0	15	18	3600.0	15	18	3600.0	3605.0	3600.0	3600.0
queen16_16	256	6320	16	23	3600.0	16	23	3600.0	16	23	3600.0	16	21	3600.0	16	20	3600.0	16	19	3600.0	3621.9	3600.0	3600.0
queen5_5	25	160	5	5	0.0	5	5	0.0	5	5	0.1	5	5	0.0	5	5	0.0	5	5	0.0	0.0	0.4	0
queen6_6	36	290	7	7	0.1	7	7	0.1	7	7	0.0	7	7	0.4	7	7	0.2	7	7	0.1	0.2	0.1	0
queen7_7	49	476	7	7	0.0	7	7	0.0	7	7	0.0	7	7	0.3	7	7	0.2	7	7	0.3	0.5	0.1	0
queen8_12	96	1368	12	12	0.0	12	12	0.1	12	12	0.1	12	12	4.5	12	12	0.4	12	12	0.4	6.1	0.3	0
queen8_8	64	728	9	9	3.4	9	9	2.8	9	9	2.6	9	9	1199.1	9	9	132.8	9	9	31.6	4.4	13.9	1
queen9_9	81	1056	10	10	13.8	10	10	159.2	10	10	921.9	9	11	3600.0	10	10	251.2	10	10	789.0	7.1	3600.0	21
r1000.1	1000	14378	20	20	0.4	20	20	0.4	20	20	0.4	20	20	44.2	20	20	7.9	20	20	5.6	0.8	0.6	0
r1000.1c	1000	485090	84	105	3600.0	83	105	3600.0	83	105	3600.0	76	0	3601.0	77	0	3600.0	75	0	3601.0	3600.0	3600.0	3600.0
r1000.5	1000	238267	234	244	3600.0	234	244	3600.0	234	234	723.1	212	0	3600.1	212	0	3600.0	212	0	3600.0	3600.0	-	3600.0
R100_1g	98	503	5	5	0.2	5	5	0.2	5	5	0.3	5	5	0.3	5	5	1.8	5	5	0.7	384.3	0.1	-
R100_5g	100	2456	12	18	3600.0	12	18	3600.0	12	18	3600.0	10	17	3600.0	12	16	3600.0	12	16	3600.0	2093.7	3600.0	-
R100_9g	100	4438	35	35	511.0	35	35	64.2	35	35	63.6	32	36	3600.1	35	35	33.8	35	35	28.5	3.5	254.2	-
r125.1	122	209	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	5	5	0.0	0.0	0.0	0
r125.1c	125	7501	46	46	0.1	46	46	0.0	46	46	0.0	46	46	0.1	46	46	0.0	46	46	0.0	0.0	0.1	0
r125.5	125	3838	36	36	0.2	36	36	0.0	36	36	0.2	33	36	3600.0	36	36	0.6	36	36	1.8	11.6	0.1	0
r250.1	250	867	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	8	8	0.0	0.0	0.1	0
r250.1c	250	30227	64	64	0.5	64	64	0.5	64	64	0.5	64	64	0.4	64	64	0.2	64	64	0.2	36.3	0.5	3
r250.5	250	14849	65	65	0.5	65	65	0.4	65	65	0.6	65	67	3600.3	65	65	2.9	65	65	2.8	175.7	2.2	2
R50_1g	41	92	3	3	0.0	3	3	0.0	3	3	0.0	3	3	0.0	3	3	0.0	3	3	0.0	0.5	0.2	-
R50_5g	50	612	10	10	0.2	10	10	0.2	10	10	0.1	10	10	7.1	10	10	8.5	10	10	2.5	0.6	0.1	-
R50_9g	50	1092	21	21	0.1	21	21	0.0	21	21	0.1	21	21	0.1	21	21	0.0	21	21	0.0	0.2	0.1	-
R75_1g	69	249	4	4	0.1	4	4	0.0	4	4	0.2	4	4	0.0	4	4	0.0	4	4	0.0	1.8	0.1	-
R75_5g	75	1407	12	12	41.2	12	12	215.8	12	12	1122.1	10	13	3600.0	11	13	3600.0	11	13	3600.0	215.1	2029.3	-
R75_9g	75	2513	31	36	3600.0	31	36	3600.0	31	36	3600.0	30	33	3600.0	33	33	194.2	33	33	135.1	0.6	3070.7	-
school1	385	19095	14	14	0.2	14	14	0.1	14	14	0.2	14	14	8.4	14	14	6.5	14	14	7.3	3623.8	0.7	0
school1_nsh	352	14612	14	14	0.2	14	14	0.1	14	14	0.1	14	14	15.5	14	14	12.6	14	14	31.9	1911.7	0.3	0
wap01a	2368	110871	41	41	2013.0	41	41	2568.7	41	47	3600.0	40	0	3600.0	40	0	3600.0	40	0	3600.0	3600.0	589.8	3600.0
wap02a	2464	111742	40	40	1097.7	40	40	2879.0	40	47	3600.0	40	0	3600.1	40	0	3600.0	40	0	3600.0	3600.0	316.4	3600.0
wap03a	4730	286722	40	55	3600.0	40	55	3600.0	40	55	3600.0	40	0	3605.4	40	0	3600.2	40	0	3600.4	3600.0	3600.0	3600.0
wap04a	5231	294902	40	49	3600.0	40	49	3600.0	40	49	3600.0	40	0	3600.0	40	0	3600.0	40	0	3600.1	3600.0	-	3600.0
wap05a	905	43081	50	50	0.5	50	50	0.7	50	50	1.1	40	50	3600.2	50	50	570.9	41	0	3600.0	4.3	1.5	1
wap06a	947	43571	40	40	64.0	40	40	340.4	40	40	380.2	40	49	3600.9	40	0	3600.0	40	47	3600.1	3600.0	40.9	3600.0
wap07a	1809	103368	40	45	3600.0	40	45	3600.0	40	45	3600.0	40	0	3600.0	40	0	3600.0	40	0	3600.0	3600.0	3600.0	3600.0
wap08a	1870	104176	40	40	461.2	40	40	457.8	40	40	1202.0	39	0	3600.1	40	0	3600.0	40	0	3600.0	3600.0	3600.0	3600.0
will199GPIA	701	6772	7	7	0.1	7	7	0.1	7	7	0.1	7	7	0.5	7	7	0.3	7	7	0.3	2.2	0.8	0
zeroin.i.1	126	4100	49	49	0.0	49	49	0.0	49	49	0.0	49	49	0.1	49	49	0.1	49	49	0.1	0.1	0.1	0
zeroin.i.2	157	3541	30	30	0.1	30	30	0.0	30	30	0.1	30	30	0.8	30	30	0.1	30	30	0.1	0.0	0.1	0
zeroin.i.3	157	3540	30	30	0.1	30	30	0.1	30	30	0.0	30	30	0.8	30	30	0.1	30	30	0.1	0.0	0.1	0
#solved					**98**			**98**			95			72			88			86	73	**98**	84

B Detailed results of the seven models for the 33 DIMACS instances for the BCP

Table 5 Results of the seven models for the 33 DIMACS instances for the BCP.

			POP-S-B			POPH-S-B			ASS-S-B			POP-I-B			POPH-I-B			ASS-I-B			DFMM		
Instance	V	E	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]	lb	ub	time[s]
GEOM20	20	20	21	21	0.0	21	21	0.0	21	21	0.0	21	21	0.1	21	21	0.1	21	21	0.1	21	21	0.0
GEOM20a	20	37	20	20	0.0	20	20	0.0	20	20	0.0	20	20	0.5	20	20	0.5	20	20	0.6	20	20	0.0
GEOM20b	20	32	13	13	0.0	13	13	0.0	13	13	0.0	13	13	0.1	13	13	0.0	13	13	0.1	13	13	0.0
GEOM30	30	50	28	28	0.0	28	28	0.0	28	28	0.1	28	28	4.8	28	28	2.6	28	28	0.5	28	28	0.1
GEOM30a	30	81	27	27	0.0	27	27	0.0	27	27	0.1	27	27	3.1	27	27	3.2	27	27	2.8	27	27	0.1
GEOM30b	30	81	26	26	0.0	26	26	0.0	26	26	0.0	26	26	1.0	26	26	2.7	26	26	0.4	26	26	0.0
GEOM40	40	78	28	28	0.0	28	28	0.1	28	28	0.1	28	28	3.6	28	28	8.4	28	28	0.8	28	28	0.1
GEOM40a	40	146	37	37	0.2	37	37	0.3	37	37	2.1	37	37	133.1	37	37	17.1	37	37	12.8	37	37	1.4
GEOM40b	40	157	33	33	0.1	33	33	0.1	33	33	1.5	33	33	9.3	33	33	240.4	33	33	13.4	33	33	2.1
GEOM50	50	127	28	28	0.1	28	28	0.1	28	28	0.2	28	28	6.6	28	28	36.7	28	28	2.3	28	28	0.3
GEOM50a	50	238	50	50	0.8	50	50	1.0	50	50	87.1	50	50	280.4	38	50	3600.0	50	50	60.8	50	50	374.4
GEOM50b	50	249	35	35	0.5	35	35	0.5	35	35	4.9	35	35	2028.1	35	35	683.9	35	35	250.6	35	35	144.7
GEOM60	60	185	33	33	0.2	33	33	0.1	33	33	0.3	33	33	30.2	33	33	56.7	33	33	3.5	33	33	1.1
GEOM60a	60	339	50	50	1.0	50	50	0.8	50	50	112.4	50	50	1124.3	38	50	3600.0	50	50	170.4	50	50	684.6
GEOM60b	60	366	41	41	2.4	41	41	1.7	41	41	29.3	34	41	3600.0	33	41	3600.0	40	42	3600.0	41	41	22915.9
GEOM70	70	267	38	38	0.1	38	38	0.1	38	38	2.3	38	38	36.0	38	38	36.6	38	38	17.6	38	38	2.4
GEOM70a	70	459	61	61	6.3	61	61	6.8	61	61	561.0	44	61	3600.0	35	62	3600.0	60	61	3600.0	61	61	24798.0
GEOM70b	70	488	47	47	4.9	47	47	5.4	47	47	146.0	34	49	3600.0	47	47	3318.3	35	50	3600.1	47	47	534.6
GEOM80	80	349	41	41	0.3	41	41	0.3	41	41	4.0	41	41	132.9	41	41	1103.3	41	41	46.4	41	41	8.2
GEOM80a	80	612	63	63	11.4	63	63	10.3	63	63	964.2	40	63	3600.0	31	64	3600.0	49	65	3600.0	63	63	87770.8
GEOM80b	80	663	60	60	9.8	60	60	8.1	60	60	894.4	29	62	3600.0	26	62	3600.0	44	66	3600.0	60	60	54320.9
GEOM90	90	441	46	46	2.0	46	46	2.0	46	46	21.5	46	46	1679.2	46	46	1607.1	46	46	70.6	46	46	55.2
GEOM90a	90	789	63	63	11.0	63	63	12.6	63	63	1447.3	29	67	3600.0	32	65	3600.0	47	69	3600.1	63	63	130050.1
GEOM90b	90	860	69	69	58.0	69	69	56.2	67	109	3600.0	27	75	3600.1	31	73	3600.0	48	85	3600.1	$-\infty$	69	172800.0
GEOM100	100	547	50	50	1.1	50	50	1.5	50	50	87.8	33	50	3600.0	41	50	3600.0	50	50	238.1	50	50	545.8
GEOM100a	100	992	66	66	185.0	66	66	316.3	65	91	3600.0	27	75	3600.0	36	72	3600.0	35	81	3600.0	$-\infty$	70	172800.0
GEOM100b	100	1050	71	71	156.6	71	71	136.3	69	121	3600.0	24	78	3600.0	22	78	3600.0	42	86	3600.1	$-\infty$	71	172800.0
GEOM110	110	638	50	50	1.7	50	50	1.7	50	50	98.9	37	50	3600.0	40	51	3600.0	50	50	319.1	50	50	2982.2
GEOM110a	110	1207	69	69	287.1	69	69	160.5	68	109	3600.0	26	77	3600.0	25	79	3600.0	43	88	3600.1	$-\infty$	73	172800.0
GEOM110b	110	1256	77	77	620.8	77	77	608.1	76	121	3600.0	21	86	3600.0	22	84	3600.0	40	95	3600.1	$-\infty$	79	172800.0
GEOM120	120	773	59	59	4.2	59	59	4.5	59	59	516.6	24	61	3600.0	43	59	3600.0	59	59	609.2	59	59	10778.2
GEOM120a	120	1434	82	82	338.2	82	82	293.0	82	109	3600.0	19	90	3600.0	28	93	3600.0	46	92	3600.1	$-\infty$	84	172800.0
GEOM120b	120	1491	81	123	3600.0	81	123	3600.0	83	123	3600.0	19	102	3600.0	28	97	3600.0	39	100	3600.1	$-\infty$	85	172800.0
#solved					**32**			**32**			26			17			16			20			26

Entailing Generalization Boosts Enumeration

Dror Fried
Department of Mathematics and Computer Science, The Open University of Israel, Ra'anana, Israel

Alexander Nadel
Intel Corporation, Israel and Faculty of Data and Decision Sciences, Technion, Haifa, Israel

Roberto Sebastiani
DISI, University of Trento, Italy

Yogev Shalmon
Intel Corporation, Israel and The Open University of Israel, Ra'anana, Israel

Abstract

Given a combinational circuit Γ with a single output o, AllSAT-CT is the problem of enumerating all solutions of Γ. Recently, we introduced several state-of-the-art AllSAT-CT algorithms based on *satisfying generalization*, which generalizes a given total Boolean solution to a smaller ternary solution that still satisfies the circuit. We implemented them in our open-source tool `HALL`. In this work we draw upon recent theoretical works suggesting that utilizing generalization algorithms, which can produce solutions that *entail* the circuit without *satisfying* it, may enhance enumeration. After considering the theory and adapting it to our needs, we enrich `HALL`'s AllSAT-CT algorithms by incorporating several newly implemented generalization schemes and additional SAT solvers. By conducting extensive experiments we show that *entailing* generalization substantially boosts `HALL`'s performance and quality (where *quality* corresponds to the number of reported generalized solutions per instance), with the best results achieved by combining *satisfying* and entailing generalization.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases Generalization, Minimization, Prime Implicant, AllSAT, SAT, Circuits

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.13

Supplementary Material *Software*: `https://github.com/yogevshalmon/allsat-circuits`
archived at `swh:1:dir:9c4d2c70cd996ca68c8837178682cb4c32cd4e20`

Funding We acknowledge the support of the MUR PNRR project FAIR – Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU. The work was partially supported by the project "AI@TN" funded by the Autonomous Province of Trento. Also funded in part by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HaDEA). Neither the European Union nor the granting authority can be held responsible for them. Grant Agreement no. 101120763 - TANGO.
Yogev Shalmon: This research was supported by The Open University of Israel's Research Fund.

Acknowledgements We are grateful to Ben Emanuel for helpful discussions which played an important role in shaping our research. Also, we thank the anonymous reviewers for their comments and useful suggestions.

1 Introduction

Enumerating the solutions of a given propositional formula is often a required task in computer science [20, 11, 22, 49, 52]. In *AllSAT-CT*, the formula is provided in a form of a combinational circuit $\Gamma = \langle I, G, o\rangle$ with inputs I, gates G and a single output o. Then, the goal is to enumerate all the possible assignments to Γ's inputs, for which Γ's output is 1 (see Fig. 1 for an example). AllSAT-CT's applications include model checking [28, 17, 18] and

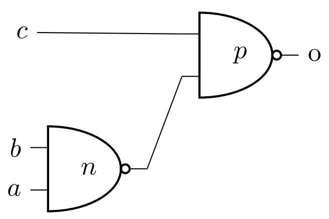
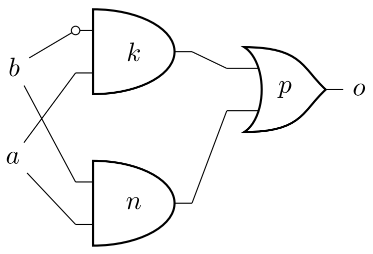


Figure 1 The circuit $\Gamma = \langle I = \{a, b, c\}, G = \{n \leftrightarrow a \wedge b, p \leftrightarrow \neg n \wedge c\}, o \equiv \neg p\rangle$ is shown. An AllSAT-CT solver could return the following two solutions: $\sigma_1 \equiv \{c := 0\}$ and $\sigma_2 \equiv \{a := 1; b := 1\}$.



(a) $\Gamma = \langle I = \{a, b\}, G = \{n \leftrightarrow a \wedge b, k \leftrightarrow a \wedge \neg b, p \leftrightarrow n \vee k\}, o \equiv p\rangle$.

$$C_1 = (n \vee \neg a \vee \neg b),$$
$$C_2 = (\neg n \vee a), C_3 = (\neg n \vee b),$$
$$C_4 = (k \vee \neg a \vee b),$$
$$C_5 = (\neg k \vee a), C_6 = (\neg k \vee \neg b),$$
$$C_7 = (\neg p \vee k \vee n),$$
$$C_8 = (p \vee \neg k), C_9 = (p \vee \neg n)$$

(b) Encoding G to CNF.

Figure 2 An example where UC generalization returns an e-hard solution (that is, an e-generalization which cannot be subsumed by any s-generalization). Let $\sigma \equiv \{a := 1, b := 1\}$ be the solution to Γ, depicted in Fig. 2a, we are interested to generalize to $\tau \equiv \{a := 1\}$. τ is an e-generalization of σ, since with $a = 1$ the output must be 1 whether b is assigned 1 or 0, but not an s-generalization, since ternary simulating τ would assign X to k, n and o. Clearly, τ is also not subsumed by any other solution, so τ is e-hard. The translation of $\neg\Gamma$ to CNF using Tseitin encoding would contain the clauses in Fig. 2b and the unit clause $(\neg p)$ representing the negation of the output. Propagating $\neg p$ by the SAT solver would imply $\neg n$ and $\neg k$ in the clauses C_8 and C_9. One can now see that assuming $a = 1$ is sufficient to get a conflict between C_1 and C_4, hence the unit cube $Q = a$ could potentially be returned by the solver as the UC, which induces $\tau^Q \equiv \{a := 1\}$ as required.

ATPG [9, 48]. Moreover, we apply AllSAT-CT solving in our industrial practice for Static Timing Analysis (STA) [46, 14], which is a crucial step in circuit design that validates the timing of a circuit by checking all possible paths for timing violations.

In a recent work of ours [14], we have introduced several anytime AllSAT-CT algorithms that work by iteratively retrieving a solution, generalizing it, reporting it to the user, and subsequently blocking it. These algorithms, implemented into an open-source tool called `HALL`, exhibited state-of-the-art performance and quality (where *quality* corresponds to the number of reported generalized solutions). Increasing the quality is vital in AllSAT-CT, particularly in STA, where testing as few potential timing violations as possible is required. In this work, we have substantially improved both performance and quality of `HALL` on a wide range of benchmarks, mainly by upgrading `HALL`'s generalization component, leveraging the insights outlined below.

We first discuss generalization. Given a circuit Γ and its total Boolean solution $\sigma(I)$, it is often required to *generalize* σ to a small *ternary* solution by replacing as many Boolean values as possible by X's (don't cares), while making sure that the generalized σ is still a solution to Γ. Generalization is a variant of (prime) implicant generation, the latter extensively studied since the 1950th [39, 27, 44, 19, 7, 10, 38, 23], where in generalization there is a starting solution that must be subsumed by the resulting implicant. Since the early 2010s, generalization has been widely used as a core component in IC3 (aka PDR) model checking algorithm and its derivatives [6, 8, 12, 51, 21]. A careful look, however, reveals that the definition of generalization is ambiguous. Indeed, since generalization generates ternary assignments, to define it one must answer the following question: what does it mean for a given ternary input assignment $\tau(I) : I \mapsto \{0, 1, X\}$ to serve as a solution to the circuit? One possibility would be as follows. Every $\tau(I)$ can be expanded to the assignment $\tau^S(I \cup G \cup \{o\})$ by propagating $\tau(I)$ to every gate and the output by *ternary simulation* (see Sect. 2). We then say that τ *satisfies* Γ (denoted by $\tau \approx \Gamma$), if $\tau^S(o) = 1$. For example, in Fig. 1, we have $\{c := 0\} \approx \Gamma$ and $\{a := 1; b := 1\} \approx \Gamma$ (assuming any omitted variables in ternary assignments are assigned X). One could have defined that a ternary τ is a solution to Γ iff $\tau \approx \Gamma$. Another option, however, inequivalent to satisfaction, is to define a ternary τ to comprise a solution to Γ iff τ *entails* the circuit, where τ *entails* Γ (denoted by $\tau \models \Gamma$), if $\rho^S(o) = 1$ for *any* ρ which substitutes *every* X in τ by *any* Boolean value. To understand why entailment is preferable to satisfaction for solution definition, consider the circuit Γ in Fig. 2a (for now ignore Fig. 2's caption, discussed in Sect. 3) and the assignment $\psi \equiv \{a := 1\}$. ψ qualifies as a solution to Γ through both intuitive understanding and our entailment-based solution definition, since ternary simulation renders either $k = 1$ or $n = 1$ for either $b = 0$ or $b = 1$, respectively, so $o = 1$ is implied no matter what. However, ψ does *not* satisfy Γ, since, given $b = X$, ternary simulation would assign X to both the gates k and n and then the output too.

The core of our analysis is based on our previously unpublished work [42] and later follow-ups [30, 31], which made the key distinction between entailment and satisfaction and surmised that integrating duality [15, 29]-based generalization algorithms, expected to output solutions which entail the formula without satisfying it, should boost enumeration. Our work, however, is the first to exhibit how to capitalize on this observation to advance the state of the art in enumeration empirically, thereby bridging the gap between theory and practice (the duality-based model counter `dualiza` [29] can also solve AllSAT-CT, but Sect. 5 shows that it is inefficient).

As such, we present in Sect. 3 three distinct generalization definitions, ordered in a hierarchy, including the most powerful *entailing (e-)generalization* where the generalized τ has to merely entail Γ, followed by *satisfying (s-)generalization* where τ must satisfy Γ, itself followed by an even more restricted *gate (g-)generalization* (intended to argue that generalizing after reducing the circuit to clauses is inefficient). The first two definitions are based on our previously unpublished work [42], while the third one is novel. We then classify commonly used generalization algorithms based on our hierarchy and observe that duality-based *Unsatisfiable Core-based (UC) generalization* [8] can potentially actualize the advantage of e-generalization.

Next, we leverage our analysis to boost `HALL`, so far based on (forward) *ternary generalization* [41, 12], restricted to s-generalization. Substituting ternary by UC generalization substantially improves `HALL`'s performance and quality, further improved by combining ternary and UC generalization (as UC generalization does not guarantee the smallest cardinality). Additionally, we study and compare the impact of the following newly implemented components in `HALL`: the SAT solvers `CaDiCaL` [4], `MergeSat` [25] and `CryptoMiniSAT` [45] (added alongside `IntelSAT` [35]), *backward* ternary generalization [41] (aka *justification* [43]) and UC generalization [8] with or without minimization [40, 32]. In what follows, Sect. 4 discusses AllSAT-CT. Sect. 5 is dedicated to experimental evaluation. In Sect. 6 we conclude.

2 Preliminaries

We briefly review the relevant syntax of Boolean logic. Let V be the set of Boolean variables. A *literal* l is either a variable $v \in V$ or its negation $\neg v$. A *clause/cube* is a disjunction/conjunction of literals. A formula $F(V)$ is in *Conjunctive/Disjunctive Normal Form (CNF/DNF)* if it is a conjunction/disjunction of clauses/cubes. A (combinational Boolean single-output) *circuit* $\Gamma = \langle I = \{v_1, \cdots v_n\}, G = \{v_{n+1}, \cdots v_{n+m}\}, o \in \{v_{n+m}, \neg v_{n+m}\}\rangle$ is a tuple, where I are the *inputs*, G are the *gates* and o is the *output*. Every gate comprises the formula $v_k \leftrightarrow (l_i \wedge l_j)$, where $i, j < k$ and l_i, l_j are literals of variables v_i and v_j respectively (using only $\wedge$ operator does not restrict the generality [5]). Tseitin encoding [50] converts a given circuit Γ to a CNF formula by translating every gate $v \leftrightarrow l_1 \wedge l_2$ to three clauses $(v \vee \neg l_1 \vee \neg l_2) \wedge (\neg v \vee l_1) \wedge (\neg v \vee l_2)$ and adding the unit clause (o) to assert the output.

For brevity, we skip the standard Boolean logic semantics. *Ternary logic* [37] extends Boolean logic with an additional value called *don't-care (X)*. Formally, a *ternary* assignment $\tau : V \mapsto \{0, 1, X\}$ assigns each variable to one of the *ternary values* $\{0, 1, X\}$. The *cardinality* $|\tau|$ of a ternary assignment τ is the number of variables in τ assigned 0 or 1 (inducing an order relation between assignments). A ternary assignment is also a *total Boolean assignment* iff it has the maximal cardinality. To evaluate a formula in Boolean logic syntax under a ternary assignment τ, one can use Boolean logic semantics extended by the rules $(\neg X \equiv X)$, $(X \wedge 1 \equiv X)$, $(X \wedge 0 \equiv 0)$ and $(X \wedge X \equiv X)$.

Ternary simulation propagates a given ternary assignment to the inputs τ across the gates all the way to the output:

▶ **Definition 1** (Ternary Simulation [41, 16]). *Given a circuit* $\Gamma = \langle I, G, o\rangle$ *and a ternary assignment* $\tau(I) : I \mapsto \{0, 1, X\}$ *to* Γ*'s inputs,* ternary simulation *transforms* τ *to the assignment* $\tau^S(\{v_1 \dots v_{n+m}\})$*, where* $\tau^S(v) := \tau(v)$ *for every input* $v \in I$*, and for every gate* $v_k \leftrightarrow (l_i \wedge l_j)$*, we have* $\tau^S(v_k) := \tau^S(l_i) \wedge \tau^S(l_j)$*.*

For brevity, we omit variables assigned X when specifying ternary assignments. We say that a ternary assignment $\rho(I)$ *subsumes* the ternary assignment $\tau(I)$, denoted by $\rho \subseteq \tau$, if $\tau(v) = \rho(v)$ for every $v : \rho(v) \in \{0, 1\}$. We say that $\rho(I)$ *strictly subsumes* $\tau(I)$, denoted by $\rho \subset \tau$, if $\rho \subseteq \tau$ and $|\rho| < |\tau|$. For example, $\{x_1 := 1\} \subset \{x_1 := 1, x_2 := 0\}$. A ternary assignment $\tau(I)$ naturally induces the cube D^τ containing v wherever $\tau(v) = 1$ and $\neg v$ wherever $\tau(v) = 0$ (variables assigned X's are skipped). Similarly, a cube $D(I)$ induces a ternary assignment, denoted by τ^D, in which $\tau(v) = 1$ for $v \in D$, $\tau(v) = 0$ for $\neg v \in D$ and $\tau(v) = X$ if $v, \neg v \notin D$. For example, given $I = \{a, b, c\}$, $\tau(I) \equiv \{a := 1, b := 0\}$ induces the cube $D^\tau = a \wedge \neg b$, while the cube $D(I) = a \wedge \neg b$ induces $\tau^D \equiv \{a := 1, b := 0\}$.

Given a CNF formula F, a SAT solver decides whether F is satisfiable. Many SAT solvers are *incremental* [13, 36]: they can be invoked multiple times, where, for every new query SAT(F, A), the SAT solver also receives a cube of *assumption literals (assumptions)* A, which hold only for the current query. The solver then decides whether $F \wedge A$ is satisfiable (where F contains all the clauses provided so far). If $F \wedge A$ is unsatisfiable, SAT(F, A) returns an *Unsatisfiable Core (UC)*, that is, a cube $A' \subseteq A$, such that $F \wedge A'$ is still unsatisfiable [13].

3 The Generalization Hierarchy

Recall from Sect. 1 the following definitions of a ternary assignment $\tau(I) : I \mapsto \{0, 1, X\}$ *satisfying* ($\approx\!\!|$) and *entailing* ($\models$) a given circuit $\Gamma = \langle I, G, o\rangle$:

1. τ *satisfies* Γ (denoted by $\tau \approx\!\!| \Gamma$), if $\tau^S(o) = 1$,
2. τ *entails* Γ (denoted by $\tau \models \Gamma$), if $\rho^S(o) = 1$ for *any* ρ which substitutes *every* X in τ by *any* Boolean value.

We define a solution to the least restrictive option sufficient for real-world applications (e.g., AllSAT-CT or PDR): $\tau(I)$ is a *solution* to Γ iff $\tau \models \Gamma$.

In addition, we say that τ *satisfies the gate* $v \in G$, if $\tau^S(v) \neq X$, and that τ *gate-satisfies* Γ if τ satisfies Γ and every gate in Γ. Def. 2 offers three alternatives for defining generalization, where *any* generalization τ must subsume the given total Boolean solution σ.

▶ **Definition 2** (G-,s-,e-generalization). *Given a circuit* $\Gamma = \langle I, G, o\rangle$ *and its total Boolean solution* $\sigma(I) {\approx\!\!\!\!|}\Gamma$*, a ternary solution* $\tau(I) \models \Gamma : \tau(I) \subseteq \sigma(I)$ *is a:*

- gate (g-) generalization *of* σ *if* τ *gate-satisfies* Γ *(that is,* $\tau {\approx\!\!\!\!|}\Gamma$ *and* $\forall v \in G : \tau^S(v) \neq X$*)*
- satisfying (s-) generalization *of* σ *if* $\tau {\approx\!\!\!\!|}\Gamma$
- entailing (e-) generalization *of* σ *if* $\tau \models \Gamma$

E-generalization is the least restrictive one, merely requiring τ to be Γ's solution. S-generalization requires τ to satisfy the circuit, while g-generalization additionally has τ satisfying every single gate. We denote the sets of all the g-, s- and e- generalizations for a given circuit Γ and a total Boolean solution $\sigma {\approx\!\!\!\!|}\Gamma$ by $G(\Gamma, \sigma)$, $S(\Gamma, \sigma)$ and $E(\Gamma, \sigma)$, respectively. Towards separating between g- and s-generalization as well as between s- and e-generalization, Def. 3 introduces the notions of s-hard and e-hard solutions.

▶ **Definition 3** (S-hard, e-hard). *Given a circuit* $\Gamma = \langle I, G, o\rangle$ *and its total Boolean solution* $\sigma(I) {\approx\!\!\!\!|}\Gamma$*, a ternary solution* $\tau(I) \models \Gamma$ *is*

- s-hard *if* $\tau \in S(\Gamma, \sigma)$*, but for every* $\rho \subseteq \tau : \rho \notin G(\Gamma, \sigma)$
- e-hard *if* $\tau \in E(\Gamma, \sigma)$*, but for every* $\rho \subseteq \tau : \rho \notin S(\Gamma, \sigma)$

Lemma 4 below presents the generalization hierarchy. It shows that e-generalization is *more powerful (denoted by* $\gg$*)* than s-generalization in the following sense: every s-generalization is an e-generalization, but there exists an e-hard solution τ which separates between e- and s-generalization (that is, τ is an e-generalization, but no $\rho \subseteq \tau$ is an s-generalization). Similarly, s-generalization $\gg$ g-generalization. The generalization hierarchy is illustrated in Fig. 3.

▶ **Lemma 4** (e-generalization $\gg$ s-generalization $\gg$ g-generalization). *The lemma is threefold:*

I. *For every* $\Gamma = \langle I, G, o\rangle$ *and total Boolean* $\sigma(I) {\approx\!\!\!\!|}\Gamma$*:* $G(\Gamma, \sigma) \subseteq S(\Gamma, \sigma) \subseteq E(\Gamma, \sigma)$*.*
II. *There exists an s-hard solution for some* $\Gamma = \langle I, G, o\rangle$ *and a total Boolean* $\sigma(I) {\approx\!\!\!\!|}\Gamma$*.*
III. *There exists an e-hard solution for some* $\Gamma = \langle I, G, o\rangle$ *and a total Boolean* $\sigma(I) {\approx\!\!\!\!|}\Gamma$*.*

Proof. I is straightforward. For II, consider Fig. 1, where $\{c := 0\}$ is s-hard, given $\sigma = \{a := 1; b := 1; c := 0\}$ (since the output is satisfied by ternary simulating τ, but gate n is not). For III, consider Fig. 2a, where $\{a := 1\}$ is e-hard, given $\sigma = \{a := 1, b := 1\}$. ◀

Next, we classify popular generalization algorithms, based on the hierarchy in Lemma 4. Consider any *Tseitin generalization* algorithm which translates the circuit to a CNF using Tseitin encoding and generalizes at CNF level by any algorithm (see, e.g. [7, 10, 49]) that turns as many variables as possible to don't cares, while still guaranteeing that every clause is satisfied. Such algorithms can only generate g-generalizations. Indeed, in Tseitin encoding, every gate $v \leftrightarrow l_1 \wedge l_2$ is translated to $(v \vee \neg l_1 \vee \neg l_2) \wedge (\neg v \vee l_1) \wedge (\neg v \vee l_2)$. Hence, the variable v representing the gate must be assigned a Boolean value, since, otherwise, one or two of the three clauses (depending on the values of l_1 and l_2) would have been left unsatisfied.

Let (forward) *ternary generalization* [41, 12] be the algorithm that generalizes a given solution by iteratively assigning every input v to X iff propagating $v := X$ by ternary simulation still sets the output to 1. While, in principle, the inputs can be visited in any order, our implementation visits the inputs in their order from 1 to n.

Note that ternary generalization can also be carried out *backwards* [41], where *backward ternary generalization* is also known as *justification* [43]. Briefly speaking, backward ternary generalization traverses the circuit's gates in a reversed order (starting from the output).

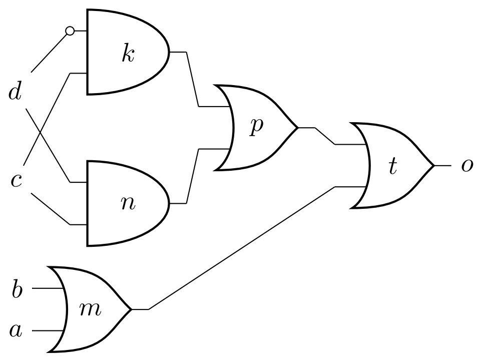


Figure 3 Illustrating the generalization hierarchy-related concepts on the circuit $\Gamma = \langle I = \{a, b, c, d\}, G = \{m \leftrightarrow a \vee b, n \leftrightarrow c \wedge d, k \leftrightarrow c \wedge \neg d, p \leftrightarrow n \vee k, t \leftrightarrow p \vee m\}, o \equiv t\rangle$. All of the following assignments are solutions to Γ: $\sigma \equiv \{a := 1, b := 1\, c := 1, d := 1\}$, $\rho \equiv \{b := 1, c := 1, d := 1\}$, $\tau \equiv \{b := 1, c := 1\}$ and $\mu \equiv \{c := 1\}$, where σ is the only Boolean solution, and we have $\mu \subset \tau \subset \rho \subset \sigma$ by construction. Observe that μ is an e-hard e-generalization of σ, τ is an s-hard s-generalization of σ, whereas ρ is a g-generalization of σ (but ρ is not an s- nor an e-generalization).

Whenever a gate whose output is not X is encountered, the algorithm tries to convert one of its inputs to X, whenever possible (e.g., for an $\wedge$-gate, whose output and inputs are all 0, one of the inputs can be converted to X).

Ternary generalization (both forward and backward) can generate s-hard solutions (e.g., it could generalize $\{a := 1; b := 1; c := 0\}$ to $\{c := 0\}$ in Fig. 1), but not e-hard solutions, since it uses ternary simulation for establishing satisfiability. Same holds for *dual-rail generalization* [14, 43], which applies generalization at CNF level but using ternary-logic-simulating dual-rail encoding. Specifically, in dual-rail encoding, every variable v in the original circuit is mapped to two Boolean *dual-rail* variables (v^+, v^-) in the resulting CNF, where assigning both v^+ and v^- to 0 corresponds to assigning the original v to a don't-care. Then, one can guide the SAT solver to return a generalized solution by applying anytime MaxSAT-inspired heuristics [33, 34] to increase the number of don't-cares assigned to the circuit inputs (that is, the number of 0's assigned to their respective dual-rail variables). In line with our analysis, state-of-the-art AllSAT-CT algorithms are substantially faster with ternary or dual-rail generalization than with Tseitin generalization [14].

Finally, recall UC generalization [8] (its predecessors being implication graph-based approaches [47, 28, 40]). Given a circuit $\Gamma = \langle I, G, o\rangle$, let the *dual circuit* $\neg\Gamma$ be $\langle I, G, \neg o\rangle$. Let $\sigma(I) : I \mapsto \{0, 1\}$ be Γ's total Boolean solution. Note that σ does *not* satisfy $\neg\Gamma$. Let $\neg F$ be a conversion of $\neg\Gamma$ to CNF using Tseitin encoding. *Unsatisfiable Core-based (UC) generalization* [8] generalizes $\sigma(I)$ to τ^Q, where Q is the unsatisfiable core (cube), returned by the query $\mathrm{SAT}(\neg F, D^\sigma)$. For example, $\sigma \equiv \{a := 1, b := 1\}$ is a solution to Γ in Fig. 2a, hence $\mathrm{SAT}(\neg F, a \wedge b)$ must return UNSAT, and the example UC a would translate to $\tau \equiv \{a := 1\}$, which generalizes σ. UC generalization guarantees e-generalization as substituting X's in τ^Q by any Boolean values and ternary simulating must render $o = 1$, otherwise $Q \wedge \neg F$ would have been satisfiable. Crucially, unlike the other algorithms, UC generalization can generate e-hard solutions: see Fig. 2 for a detailed example. One can also minimize the UC [40, 32].

4 Generalization-based Enumeration Algorithms

Given a circuit $\Gamma = \langle I, G, o\rangle$, an AllSAT-CT solver returns a DNF formula $Q(I)$, where for every *solution cube* $D(I) \in Q(I)$, we have $\tau^D \models \Gamma$, while $G \wedge o$ and $Q(I)$ are logically equivalent. We next review the AllSAT-CT algorithms from [14] and introduce our new UC generalization-based algorithms CORE, ROC and CARMA. All the algorithms are implemented within the well-known *blocking* framework, which repeatedly enumerates, generalizes and blocks the solutions [28] (a correctness proof can be found in [28]). This work focuses on non-disjoint solving (i.e., a total Boolean solution can be subsumed by multiple solutions), since disjoint solving [52], although supported by HALL, would be impractical for AllSAT-CT applications in model checking [28, 17, 18], ATPG [9, 48] and STA [46, 14]. As a side note, AllSAT-CT is simpler than finding *all* the prime implicants [44, 38, 23], since we only need a *subset* of the (not-necessarily-prime) implicants which subsume every total Boolean solution.

Consider Alg. 1 that presents TALE from [14] and our novel CORE and ROC algorithms. To recall TALE let us follow Alg. 1 in TALE mode (A = TALE). First, the algorithm converts the given circuit to CNF by applying the Tseitin encoding and provides the CNF as an input to an incremental SAT solver instance plain (line 1). Line 3 initializes the DNF Q that will contain all the solutions. Then, the algorithm starts to iteratively produce cubes in the following way. It queries plain to get a total Boolean solution $\sigma \models\!\approx \Gamma$ (line 5), applies ternary generalization (the forward version by default) to generalize σ (line 6), updates DNF Q with the cube U induced by σ (lines 8 and 12) and blocks U in plain (line 13).

Algorithm 1 Three AllSAT-CT algorithms: TALE, CORE and ROC

```
    Input: Circuit $\Gamma = \langle I, G, o\rangle$        Input: A $\in$ {TALE, CORE, ROC}
    Output: DNF $Q(I)$
 1: plain := CNFTseitin($\Gamma$)                                  ▷ Initialize plain SAT instance
 2: if A $\neq$ TALE then dual := CNFTseitin($\neg\Gamma$)        ▷ Initialize dual SAT instance, if required
 3: $Q$ := {}          ▷ Initializing the DNF $Q$, which will contain all the solutions, to be empty
 4: while not UNSAT(plain) do
 5:     $\sigma$ := SAT(plain)
 6:     if A $\neq$ CORE then $\sigma$ := TernaryGeneralize($\sigma, \Gamma$)           ▷ Ternary generalization
 7:     if A = TALE then
 8:         $U := D^\sigma$
 9:     else
10:         $U$ := SAT(dual, $D^\sigma$)                                          ▷ Fetch the UC
11:         forall $a \in U$: if SAT(dual, $U \setminus \{a\}$) is UNSAT then $U := U \setminus \{a\}$   ▷ Minimize the UC
12:     $Q := Q \vee U$                                             ▷ $Q$ is updated by the cube $U$
13:     plain := plain $\wedge \neg U$                                     ▷ Blocking $U$ in plain
14: return $Q$                                           ▷ $Q$ is not guaranteed to be disjoint
```

We now introduce our first new algorithm CORE which aims at generating e-hard solutions by switching from ternary to UC generalization. To that end, CORE uses a second incremental SAT instance dual, initialized by converting the dual circuit $\neg\Gamma$ to CNF (line 2). Then, instead of applying ternary generalization at line 6, CORE queries dual under the assumptions D^σ (line 10), that is, the cube induced by σ, to get an unsatisfiable core (cube) U, followed by iteratively minimizing it (line 11). Then, similarly to TALE, Q is updated and U is blocked.

Alg. 1 also shows another novel algorithm ROC, which applies ternary generalization (line 6), followed by UC generalization with minimization (lines 10 and 11). Despite the overhead, ROC often succeeds in generating smaller solutions than CORE, which ultimately leads to a reduction in the number of returned solutions. Generalizing further might still be possible, since our UC extraction algorithm might return a local minimum. However, finding the smallest UC would have been extremely costly [24].

Note that the initial dual invocation in CORE (line 10) is not expected to encounter any conflicts during SAT solving. This is because the assumptions represent a total Boolean input assignment, whose propagation by Boolean Constraint Propagation (BCP) must trigger a conflict with the clause $\neg o$ prior to any decision. This is not the case during the minimization loop in CORE (line 11) and any (even the initial) dual invocation in ROC, where the assumptions might represent a *partial* assignment to the inputs. This, however, is transparent to both the user and the high-level algorithm developer.

Finally, [14] introduced two additional algorithms: `MARS` based on dual-rail generalization, and `DUTY`, which combines between `TALE` and `MARS`. To test the impact of combining dual-rail and UC generalization, our third new algorithm `CARMA` upgrades `MARS` by switching to UC generalization as follows. `CARMA` is similar to `CORE`, but it uses dual-rail encoding and on-the-fly minimization [14] for `plain`, while still using Tseitin encoding for `dual`.

5 Experimental Results

We implemented our new algorithms `CORE`, `ROC` and `CARMA` in `HALL` [14] and compared them to the already implemented `TALE`, `MARS` and `DUTY`, where the default `HALL` uses `IntelSAT` [35] SAT solver for `plain` and `CaDiCaL` [4] for `dual` across all the algorithms (we provide an empirical justification for the default solver selection later in this section). We also ran the duality-based model counter `dualiza` [29] in its two enumeration modes: `sat` and `bdd`.

We used benchmarks from [14] and [26] with some further extensions. All in all, as reported below, we had started with 14 benchmark families, 10 benchmarks in each, and then removed benchmarks solved by none of the solvers in four hours, which left us with 97 benchmarks overall. We transformed each circuit family with multiple outputs to three one-output families (which AllSAT-CT solvers can handle) as follows: we applied either *or* (_or suffix below) or *xor* (_xor suffix below) operator over all the outputs similarly to [14] to create the first two one-output families, and took only the last output to create the last one-output family (_only_last_out suffix below). Below, we list all the benchmark families; the number of instances from each family solved by at least one solver appears in parenthesis:

- **random_control_or (9)**, **random_control_xor (6)** and **random_control_only_last_out (9)** from EPFL benchmark suite [1], used in [14].
- **arithmetic_or (10)**, **arithmetic_xor (1)** and **arithmetic_only_last_out (5)** from EPFL benchmark suite [1], also used in [14].
- **random_circuits_or (9)**, **random_circuits_xor (1)** and **random_circuits_only_last_out (7)**, generated by using aigfuzz [5] as in [14].
- **iscas85_or (10)**, **iscas85_xor (2)** and **iscas85_only_last_out (8)**, used in [26]; this set contains publicly available circuits, including sequential circuits. Since we consider only combinational circuits, we ignored the buffer commands while parsing the files.
- **sta_gen (10)** [14] – Static Timing Analysis (STA) industrial set: a parametrized benchmark family, which encapsulates a variety of real-world STA instances [14]. We removed the two smallest benchmarks resulting in a family of 10 benchmarks.
- **sta_gen_chunks (10)** – another family of STA benchmarks created by parameterizing the *size of each cube*, rather than the *number of inputs (N)*. Given the chunk size K and the constant number $N = 12289$, the formula $F(N, K)$ consists of a disjunction of subformulas $F_1(N, K)$ and $F_2(N, K)$, each comprising a DNF, conjuncted with the selector v_N or $\neg v_N$, respectively. In every DNF, the cubes have K variables and are mutually disjoint. The resulting formula looks as follows, where $j = (N - 1)/2$: $F(N) := F_1(N) \vee F_2(N)$, where $F_1(N) := ((v_1 \wedge v_2 \ldots \wedge v_k) \vee \ldots \vee (v_{j-k+1} \wedge \ldots v_{j-1} \wedge v_j)) \wedge v_N$ and $F_2(N) := ((v_{j+1} \wedge v_{j+2} \ldots \wedge v_{j+k}) \vee \ldots \vee (v_{N-K} \wedge \ldots v_{(N-2} \wedge v_{N-1})) \wedge \neg v_N$.

We used Intel® Xeon® machines with 32Gb memory and 3Ghz CPU frequency. We set the timeout to 1 hour and evaluated three criteria. First, *Solved* stands for the number of solved instances. The second one is *PAR-2 score* (similarly to SAT competitions [2]), where every solved benchmark contributes its run-time and every unsolved benchmark contributes twice the timeout. The lower the PAR-2 score, the better. The third criterion is *Quality*: the size (number of cubes) of the DNF, where we compared solvers by their normalized average quality, the *quality* per instance being best-known-DNF-size / current-DNF-size and 0 for unsolved instances (similarly to anytime categories at MaxSAT Evaluations [3]). The quality must be within the interval $[0, 1]$, where the higher the quality, the better.

Table 1 Our results (sorted by PAR-2 scores). The best results in each column are highlighted.

Algorithm	Origin	PAR-2	Solved	Quality
`ROC`	new	24926.845	94	0.966614
`CORE`	new	25938.178	94	0.888548
`CARMA`	new	27073.553	94	0.886870
`TALE`	[14]	92676.526	85	0.568704
`DUTY`	[14]	99447.688	84	0.560391
`MARS`	[14]	198771.013	70	0.412807
`dualiza_sat`	[29]	263810.947	61	0.459656
`dualiza_bdd`	[29]	332953.818	51	0.397921

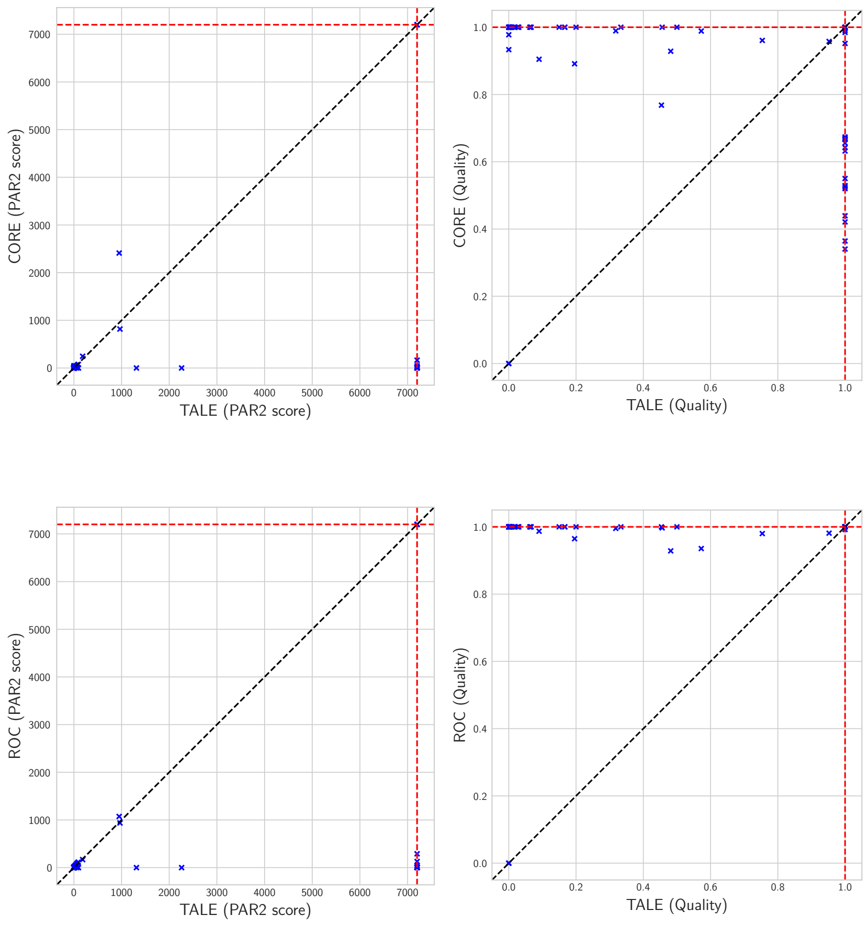


Figure 4 Comparing PAR2 score and quality: `TALE` to `CORE` (top) and `TALE` to `ROC` (bottom)

Table 1 summarizes the main results. `ROC` is the best algorithm by every criterion. It substantially outperforms the previous state-of-the-art (`TALE`), where the gap in quality is especially significant. For an instance-by-instance analysis, consider Fig. 4 starting with its

Table 2 Comparing `TALE` configurations (left) and `CORE` configurations (right).

`TALE` Config.	Solved	Quality
Default `TALE`	85	0.568704
`plain` := `CaDiCaL`	83	0.560356
`plain` := `CryptoMS`	82	0.555761
`plain` := `MergeSat`	80	0.561304
`Backward-TerSim`	72	0.501580

`CORE` Config.	Solved	Quality
Default `CORE`	94	0.888548
`dual` := `CryptoMS`	93	0.846291
`dual` := `IntelSAT`	92	0.872976
`dual` := `MergeSat`	91	0.824571
`No-UC-Minimization`	91	0.707390

upper part, which compares `TALE` to `CORE`. `CORE` is almost always on-par or better in terms of PAR-2 score, but not so in terms of quality. Consider now the lower part of Fig. 4, which compares `TALE` to `ROC`. Unlike `CORE`, `ROC` is either better or on-par with `TALE` in terms of quality on every single instance, whereas `ROC` often yields a substantially better quality. `ROC` is also always on-par or better than `TALE` in terms of PAR-2 score.

Finally, Table. 2 explains our choice of four of `HALL`'s default components. The comparison of `TALE` configurations on the left shows why we set the `plain` SAT solver default to `IntelSAT`, as `IntelSAT` outperforms `CaDiCaL` [4], `MergeSat` [25], and `CryptoMiniSAT` [45] (`CryptoMS` in Table. 2). This result is not surprising as `IntelSAT` was specifically optimized for rapid incremental mostly satisfiable queries [35]. The comparison on the left also explains why we decided against migrating from the default forward ternary generalization to the *backward* one (recall Sect. 3). The right-side table compares `CORE` configurations, supporting the choice of `CaDiCaL` as the default SAT solver for `dual` (notably, in `dual`, unlike in `plain`, the SAT queries are unsatisfiable) and the default inclusion of minimization in UC extraction.

6 Conclusion and Future Work

In this work we substantially improved the state of the art in AllSAT-CT solving in terms of both performance and quality by taking advantage of UC generalization, which can potentially yield solutions that entail the circuit without satisfying it.

Our best-performing algorithm, `ROC`, combines ternary and UC generalization as follows: it iteratively searches for solutions in an `IntelSAT`-based SAT instance `plain`. Then, every solution is generalized using forward ternary generalization, followed by further generalization to its (locally) minimal unsatisfiable core in a `CaDiCaL`-based SAT instance `dual` representing the dual circuit. The generalized solution is then reported to the user and blocked in `plain`. All the algorithms have been implemented in our open-source AllSAT-CT tool `HALL`.

Our results can be relevant for advancing disjoint AllSAT-CT solving and prime implicant enumeration [23]. Furthermore, porting our findings to model checking algorithms such as PDR [6], AVY [51], and CAR [21] could be promising. Notably, while [43] thoroughly compares different generalization approaches within PDR, it surprisingly does not conclude that UC generalization enhances PDR's performance, leaving room for potential improvement.

References

1 Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.

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

3 Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen, editors. *MaxSAT Evaluation 2023: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

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

5 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

6 Aaron R. Bradley. Sat-based model checking without unrolling. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. doi:10.1007/978-3-642-18275-4_7.

7 T. Castell. Computation of prime implicates and prime implicants by a variant of the davis and putnam procedure. In *Proceedings Eighth IEEE International Conference on Tools with Artificial Intelligence*, pages 428–429, 1996. doi:10.1109/TAI.1996.560739.

8 Hana Chockler, Alexander Ivrii, Arie Matsliah, Shiri Moran, and Ziv Nevo. Incremental formal verification of hardware. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 135–143. FMCAD Inc., 2011. URL: http://dl.acm.org/citation.cfm?id=2157676.

9 Alejandro Czutro, Ilia Polian, Piet Engelke, Sudhakar M. Reddy, and Bernd Becker. Dynamic compaction in sat-based ATPG. In *Proceedings of the Eighteentgh Asian Test Symposium, ATS 2009, 23-26 November 2009, Taichung, Taiwan*, pages 187–190. IEEE Computer Society, 2009. doi:10.1109/ATS.2009.31.

10 David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 46–52. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6679390/.

11 Imen Ouled Dlala, Saïd Jabbour, Lakhdar Saïs, and Boutheina Ben Yaghlane. A comparative study of SAT-based itemsets mining. In *Research and Development in Intelligent Systems XXXIII - Incorporating Applications and Innovations in Intelligent Systems XXIV. Proceedings of AI-2016.*, pages 37–52, 2016.

12 Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134, 2011.

13 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT, Proceedings*, 2003.

14 Dror Fried, Alexander Nadel, and Yogev Shalmon. AllSAT for combinational circuits. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.9.

15 Alexandra Goultiaeva and Fahiem Bacchus. Exploiting QBF duality on a circuit representation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010. doi:10.1609/AAAI.V24I1.7548.

16 James S. Jephson, Robert P. McQuarrie, and Robert E. Vogelsberg. A three-value computer design verification system. *IBM Systems Journal*, 8(3):178–188, 1969.

17 HoonSang Jin, HyoJung Han, and Fabio Somenzi. Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 2005. doi:10.1007/978-3-540-31980-1_19.

18 HoonSang Jin and Fabio Somenzi. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *Proceedings of the 42nd Design Automation Conference, DAC*, 2005.

19 Alex Kean and George K. Tsiknis. An incremental method for generating prime implicants/impicates. *J. Symb. Comput.*, 9(2):185–206, 1990. doi:10.1016/S0747-7171(08)80029-6.

20 Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT*, 2003.

21 Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, and Moshe Y. Vardi. Safety model checking with complementary approximations. In Sri Parameswaran, editor, *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, pages 95–100. IEEE, 2017. doi:10.1109/ICCAD.2017.8203765.

22 Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. Network verification in the light of program verification. *MSR, Rep*, 2013.

23 Weilin Luo, Hai Wan, Hongzhen Zhong, Ou Wei, Biqing Fang, and Xiaotong Song. An efficient two-phase method for prime compilation of non-clausal boolean formulae. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021*, pages 1–9. IEEE, 2021. doi:10.1109/ICCAD51958.2021.9643520.

24 Inês Lynce and João Marques-Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. URL: http://www.satisfiability.org/SAT04/programme/110.pdf.

25 Norbert Manthey. The mergesat solver. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 2021. doi:10.1007/978-3-030-80223-3_27.

26 Gabriele Masina, Giuseppe Spallitta, and Roberto Sebastiani. On CNF conversion for disjoint SAT enumeration. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.15.

27 E. J. McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956. doi:10.1002/j.1538-7305.1956.tb03835.x.

28 Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV, Proceedings*, 2002.

29 Sibylle Möhle and Armin Biere. Dualizing projected model counting. In *IEEE 30th International Conference on Tools with Artificial Intelligence, ICTAI*, pages 702–709, 2018. doi:10.1109/ICTAI.2018.00111.

30 Sibylle Möhle, Roberto Sebastiani, and Armin Biere. Four flavors of entailment. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 62–71. Springer, July 2020. doi:10.1007/978-3-030-51825-7_5.

31 Sibylle Möhle, Roberto Sebastiani, and Armin Biere. On enumerating short projected models. *CoRR*, abs/2110.12924, October 2021. arXiv:2110.12924.

32 Alexander Nadel. Boosting minimal unsatisfiable core extraction. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 221–229. IEEE, 2010. URL: `https://ieeexplore.ieee.org/document/5770953/`.

33 Alexander Nadel. Anytime weighted MaxSAT with improved polarity selection and bit-vector optimization. In *Formal Methods in Computer Aided Design, FMCAD, Proceedings*, pages 193–202, 2019.

34 Alexander Nadel. Polarity and variable selection heuristics for SAT-based anytime MaxSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2020.

35 Alexander Nadel. Introducing intel(r) SAT solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.SAT.2022.8`.

36 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *Theory and Applications of Satisfiability Testing - SAT, Proceedings*, 2012.

37 Emil L. Post. Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, 43, 1921.

38 Alessandro Previti, Alexey Ignatiev, António Morgado, and João Marques-Silva. Prime compilation of non-clausal formulae. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 1980–1988. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/281`.

39 W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952. `doi:10.1080/00029890.1952.11988183`.

40 Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004. `doi:10.1007/978-3-540-24730-2_3`.

41 J. Paul Roth, Willard G. Bouricius, and Peter R. Schneider. Programmed algorithms to compute tests to detect and distinguish between failures in logic circuits. *IEEE Trans. Electron. Comput.*, 16(5):567–580, 1967. `doi:10.1109/PGEC.1967.264743`.

42 Roberto Sebastiani. Are you satisfied by this partial assignment? *CoRR*, abs/2003.04225, February 2020. `arXiv:2003.04225`.

43 Tobias Seufert, Felix Winterer, Christoph Scholl, Karsten Scheibler, Tobias Paxian, and Bernd Becker. Everything you always wanted to know about generalization of proof obligations in PDR. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 42(4):1351–1364, 2023. `doi:10.1109/TCAD.2022.3198260`.

44 James R. Slagle, Chin-Liang Chang, and Richard C. T. Lee. A new algorithm for generating prime implicants. *IEEE Trans. Computers*, 19(4):304–310, 1970. `doi:10.1109/T-C.1970.222917`.

45 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. `doi:10.1007/978-3-642-02777-2_24`.

46 Robert B. Hitchcock Sr. Timing verification and the timing analysis program. In *Proceedings of the 19th Design Automation Conference, DAC*, 1982.

47 P. Tafertshofer and A. Ganz. Sat based atpg using fast justification and propagation in the implication graph. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 139–146, 1999. `doi:10.1109/ICCAD.1999.810638`.

48 Abraham Temesgen Tibebu and Görschwin Fey. Augmenting all solution SAT solving for circuits with structural information. In *21st IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2018, Budapest, Hungary, April 25-27, 2018*, pages 117–122. IEEE, 2018. `doi:10.1109/DDECS.2018.00028`.

49 Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)*, 2016.

50 Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

51 Yakir Vizel and Arie Gurfinkel. Interpolating property directed reachability. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2014. `doi:10.1007/978-3-319-08867-9_17`.

52 Yinlei Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. All-SAT using minimal blocking clauses. *Proceedings of the IEEE International Conference on VLSI Design*, pages 86–91, 2014.

Cooking String-Integer Conversions with Noodles

Vojtěch Havlena
Brno University of Technology, Czech Republic

Lukáš Holík
Brno University of Technology, Czech Republic

Ondřej Lengál
Brno University of Technology, Czech Republic

Juraj Síč
Brno University of Technology, Czech Republic

Abstract

We propose a method for efficient handling string constraints with string-integer conversions. It extends the recently introduced stabilization-based procedure for solving string (dis)equations with regular and length constraints. Our approach is to translate the conversions into a linear integer arithmetic formula, together with regular constraints and word equations. We have integrated it into the string solver Z3-NOODLER, and our experiments show that it is competitive and on some established benchmarks even several orders of magnitude faster than the state of the art.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases string solving, string conversions, SMT solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.14

Supplementary Material *Software (Source Code)*: `https://github.com/VeriFIT/z3-noodler` [34]
archived at `swh:1:dir:a3be55dc798afb9fa29b90ddb236653e5c13e275`

Funding This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, and the FIT BUT internal project FIT-S-23-8151.

1 Introduction

String constraint solving (string solving for short), has garnered significant attention in recent years, particularly due to its important role in verifying web programs. These programs are often written in string-intensive programming languages, such as PHP or JavaScript. Careless handling of strings can inadvertently expose systems to severe security vulnerabilities, such as SQL injection or cross-site scripting (XSS), both of which remain prevalent security risks [29, 28]. Since string constraints establish a general formal framework for working with strings, new applications of string solving still emerge. Notable examples include analyzing user policies within Amazon Web Services [25] and analyzing smart contracts [7].

String solvers are typically integrated into general SMT solvers as theory solvers, enabling the combination of string constraints with other SMT theories. The most mature SMT solvers supporting string constraints are CVC4/5 [8, 31] and Z3 [22]. Furthermore, the original string theory solver can be replaced, leading to string solvers such as Z3STR3RE [13], Z3STR4 [27], Z3-TRAU [3, 2], OSTRICH [16] (based on the PRINCESS SMT solver [32]), and, most recently, Z3-NOODLER [18, 17]. Except these general SMT solvers, there are some string-only solvers, such as NORN [5], KEPLER$_{22}$ [24], WOORPJE [21], or RETRO [19, 20].

In order to handle constraints occurring in real-world applications, string solvers need to support not only *basic* constraints, such as string equations, regular membership queries, and length constraints, but also *extended* constraints such as various kinds of string functions and predicates (e.g., `replace`, `indexof`, `substr`, or `prefix`) or string-integer conversions allowing to identify strings with their numeric values. These string manipulating functions are widely used in particular in the context of verification of web programs, as they correspond to the string operations in programming languages such as PHP or JavaScript.

Complex combinations of various types of basic and extended constraints, which is how they typically appear, is challenging for today's solvers. In this paper, we combine the recently introduced stabilization-based procedure for solving basic string constrains [14, 18] with a novel technique for handling string-integer conversions. The fundamental principle of the stabilization-based procedure is an iterative refinement of regular constraints of string variables, which represent all solutions of the string constraint, until a stable form is reached. In order to handle the string-integer conversion, namely `to_int`, `from_int`, `to_code`, and `from_code` predicates, our procedure performs reasoning over the stabilized regular constraints of string variables used in the conversions. The proposed procedure generates a *linear integer arithmetic* (LIA) formula concisely encoding possible numerical values from strings of a stabilized regular constraint and relate them with their particular lengths. If this LIA formula is satisfiable, so is the original constraint with conversions. To avoid exponential blow-up during the formula generation, we express languages as finite sets of intervals encoding all valid numerical values. Since different conversions may be applied on the same variable/sequence of variables, the generated formula must also relate the values of these conversions. In order to be precise, our procedure requires finiteness of stabilized languages of variables occurring in conversions. For infinite languages, we propose an underapproximation restricting the languages to strings of a particular length.

We implemented the proposed technique into Z3-Noodler [18, 17, 34], a string solver based on Z3 implementing the stabilization-based procedure, and compared our technique with other string solvers on all benchmarks from SMT-LIB containing string-integer conversions. Our experimental evaluation shows that our proposed technique is competitive and on many instances even several orders of magnitude faster than the state of the art.

2 Related work

The decidability of various fragments of string constraints have been studied for a long time. If a string constraint contains only string equations, the satisfiability problem is decidable [26]. Moreover, it is decidable even if we add regular constraints [33]. A question whether the combination of string equations and length constraints is decidable is still open. Adding string-integer conversions to string equations with regular and length constraints leads to undecidability [23]. The undecidability still holds even if we keep only the concatenation instead of full string equations [12].

The tools supporting string-integer conversions therefore implement incomplete procedures. The string solver cvc4/5 implements a derivation-based decision procedure for solving code-point conversions (conversions between characters and their numeric values), which are then used to define a universally quantified formula for handling string-integer conversions [30]. Z3-Trau implements string-integer conversions as a part of the flattening-based procedure [1]. The main differences between the approach to string-integer conversions in Z3-Trau and our proposed technique are twofold: (i) The approach of Z3-Trau is integrated to a string procedure underapproximating the variable languages by flat languages. We combine our approach with the fundamentally different stabilization-based procedure working with arbitrary precise regular languages. (ii) Z3-Trau underapproximates string-integer conversions restricting arguments having at most m digits (m is a parameter). To the contrary, our conversion approach works with precise regular languages describing all solutions and is complete for finite languages. We only underapproximate if the languages for conversion are infinite. Moreover, we underapproximate the precise languages and not the already underapproximated flat languages as is the case of Z3-Trau. More tools providing a (limited) support for string-integer conversions are Z3 [22], Z3str4 [11], and OSTRICH [16].

3 Preliminaries

Functions, strings and languages. We use $\mathbb{Z}$ to denote the set of integers and $[k_1, k_2]$, $k_1, k_2 \in \mathbb{Z}$, for the set of all integers between k_1 and k_2, including k_1 and k_2. For a set X we use id_X to denote the identity function over X. We fix a finite alphabet Σ (we denote symbols as $a, b, c, \ldots$) and for the rest of the paper, we assume that it contains symbols `'0'`, `'1'`, …, `'9'`. A *string* over Σ is a finite sequence $u = a_1 \ldots a_n$ of symbols from Σ. We use ϵ to denote the empty string. To avoid confusion with integers, we sometimes use quotes to denote strings with digits, e.g., `'42'` (we skip it if it is clear from the context). We further use $|u|$ to denote the *length* of u with $|\epsilon| = 0$. The set of all strings is denoted by Σ^*. The *concatenation* of strings u and v is denoted $u \cdot v$, or uv for short (ϵ is the neutral element). A *language* is a subset of Σ^*. The *concatenation* of two languages L_1 and L_2 is defined as $L_1 \cdot L_2 = \{u \cdot v \mid u \in L_1 \wedge v \in L_2\}$. *Bounded iteration* of a string/language x for $i \geq 0$ is defined recursively as (i) $x^0 = \{\epsilon\}$ for the case of languages and $x^0 = \epsilon$ for the case of strings, and (ii) $x^{i+1} = x^i \cdot x$. *Iteration* is then defined as $x^* = \bigcup_{i \geq 0} x^i$ and *positive iteration* is defined as $x^+ = \bigcup_{i \geq 1} x^i$. We denote regular languages using regular expressions RE with the following standard notation:

$$\mathsf{RE} ::= \emptyset \mid a \mid \epsilon \mid (\mathsf{RE}) \mid \mathsf{RE}^* \mid \mathsf{RE}\,\mathsf{RE} \mid \mathsf{RE} + \mathsf{RE} \mid \mathsf{RE}^n$$

where $a \in \Sigma$ and $n \in \mathbb{N}$. We further use RE^+ as a syntactic sugar for $\mathsf{RE}\,\mathsf{RE}^*$ and S to denote the regex $v_1 + \cdots + v_n$, where $S = \{v_1, \ldots, v_n\}$ is a finite subset of Σ^*.

String constraints. In this paper, we consider *string constraints* over alphabet Σ, string variables $\mathbb{X}$, and integer variables $\mathbb{I}$. The string variables range over Σ^* and integer variables over $\mathbb{Z}$. The syntax of a string contraint φ is given by the following grammar:

$$\begin{aligned}
\varphi &::= t_i \leq t_i \mid t_s = t_s \mid t_s \in \mathsf{RE} \mid \varphi \wedge \varphi \mid \neg\varphi \\
t_s &::= v \in \mathbb{X} \mid t_s \cdot t_s \mid \texttt{from\_code}(t_i) \mid \texttt{from\_int}(t_i) \\
t_i &::= v \in \mathbb{I} \mid k \in \mathbb{Z} \mid t_i + t_i \mid |t_s| \mid \texttt{to\_code}(t_s) \mid \texttt{to\_int}(t_s)
\end{aligned}$$

where t_s is a string term consisting of a concatenation of string variables[1] and string-integer conversions, and t_i is a *linear integer arithmetic* (LIA) term containing among usual arithmetic terms also a string-length term and integer conversions of string terms.

Semantic of the conversion functions is then given as follows[2]. The conversions `from_int`(k) and `to_int`(s) convert between strings and integers: if $k \geq 0$, `from_int`(k) is the string representation (without leading zeroes) of the number k and ϵ otherwise, while `to_int`(s) returns the *non-negative* number represented by the string s, or -1 if s does not represent a non-negative integer (with leading zeros allowed). For example, `from_int`(5) = `'5'`, `from_int`$(-6) = \epsilon$, and `to_int('a')` $= -1$. On the other hand,

[1] Note that an explicit string $u \in \Sigma^*$ can be encoded by a fresh variable x and a regular constraint $x \in u$.
[2] following the definition of the string theory of the SMT-LIB standard [10]

`from_code`(k) and `to_code`(s) convert between a Unicode symbol and its *code point*. More specifically, if s is a single symbol, then `to_code`(s) returns its code point, otherwise it returns -1; `from_code`(k) returns the symbol s whose code point is k (and ϵ if k is out of the range of Unicode symbols). For example, `to_code('0')` $= 48$ and `from_code`$(48) =$ `'0'`.

An *assignment* ν is a mapping $\nu\colon (\mathbb{X} \to \Sigma^*) \cup (\mathbb{I} \to \mathbb{Z})$. The value of a term in ν is defined as usual. An assignment ν is a *model* of an atomic constraint $t = s$ iff $\nu(t) = \nu(s)$, of $t \in \mathsf{RE}$ iff $\nu(t) \in \mathsf{RE}$, and of $t \leq s$ iff $\nu(t) \leq \nu(s)$. The definition of model is extended to Boolean combinations of constraints as usual.

We will often work with a pair (Lang, σ), where $\mathsf{Lang}\colon \mathbb{X} \to 2^{\Sigma^*}$ is a *language assignment* assigning a language to each string variable and $\sigma\colon \mathbb{X} \to \mathbb{X}^+$ is a *substitution*. We use $\sigma(\Phi)$ to denote the string constraint where every occurrence of every $x \in \mathbb{X}$ is replaced by $\sigma(x)$. We define the composition of two substitutions σ_1 and σ_2 as $\sigma_1 \circ \sigma_2 = \{x \mapsto \sigma_1(\sigma_2(x))\}$. We say that a substitution σ is *flat* if for each string variable x appearing in its image, $\sigma(x) = x$ (hence $\sigma \circ \sigma = \sigma$).

Normalization of string constraints. We assume that our decision procedure is used within a DPLL(T)-based SMT solver, which has the property that theory solvers are always given conjunctions of (possibly negated) atomic constraints (in the given theory). In the rest of the paper, we therefore assume that string constraints have this normal form, more specifically, that they are of the form $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$ where

- $\mathcal{E}$ is a conjunction of (dis)equations of concatenated *string variables*, i.e., they do not contain conversions `from_int`(t_i) and `from_code`(t_i). We can remove each such conversion by replacing it with a fresh string variable x and adding a new equation $x =$ `from_int`(t_i) (or $x =$ `from_code`(t_i)) to $\mathcal{C}$.
- $\mathcal{R}$ is a conjunction of regular constraints of the form $x \in \mathsf{RE}$ for $x \in \mathbb{X}$. We can convert any predicate $t_s \in \mathsf{RE}$ into this form by replacing it in $\mathcal{R}$ with $x \in \mathsf{RE}$ and adding $x = t_s$ to $\mathcal{E}$, where x is a fresh variable. Negated regular constraints $x \notin \mathsf{RE}$ can be rewritten into the positive form by complementing the corresponding regular language.
- $\mathcal{L}$ is a LIA constraint without conversions (length constraints are allowed). Again, conversions `to_int` and `to_code` are replaced with a fresh integer variable and a corresponding equation in $\mathcal{C}$.
- $\mathcal{C}$ is a conjunction of equations of the form $x =$ `from_code`(y), $x =$ `from_int`(y), $y =$ `to_code`(x), and $y =$ `from_code`(x) for $x \in \mathbb{X}$ and $y \in \mathbb{I}$ (if needed, the generic terms in arguments of conversions are replaced with a fresh variable and a corresponding equation).

Note that in the rest of the paper, we sometimes treat a conjunction of literals (i.e., atomic constraints or their negations) as a set of these literals.

4 Stabilization-based Procedure for String Constraints

We briefly describe the stabilization-based procedure from [14, 18] for solving a *conversions-free* string constraint $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ where $\mathcal{E}$ contains only *equations* (no disequations). From a high-level point of view, the stabilization-based procedure iteratively splits equations (where the equation splits are represented by substitution maps) and refines the language assignments until a stable solution is found, which is then used to generate a LIA formula describing lengths of all solutions that this language assignment represents.

Stable solution. We say that (Lang, σ), a pair of a language assignment and a substitution, is a *stable solution* of $\mathcal{E} \wedge \mathcal{R}$ if each language in Lang is nonempty, σ is flat, and every assignment $\nu\colon \mathbb{X} \to \Sigma^*$, such that $\nu(x) \in \mathsf{Lang}(x)$ and $\nu(x) = \nu(\sigma(x))$ holds for all $x \in \mathbb{X}$, is a model of $\mathcal{E} \wedge \mathcal{R}$. Loosely speaking, the stability means that we can choose arbitrary assignments of the substituted variables from Lang and, using σ, construct a model of the original constraint.

▶ **Example 1.** Consider a string constraint $xy = z \wedge w = xx \wedge w \in a^+$ over the alphabet $\{a, b\}$. Let v_1, v_2 be fresh variables, Lang a language assignment where $\mathsf{Lang}(v_1) = \mathsf{Lang}(w) = a^+$ and all other variables are mapped to $(a + b)^*$, and let $\sigma = \{x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1v_2, y \mapsto v_2, v_1 \mapsto v_1, v_2 \mapsto v_2\}$ a substitution. Note that σ is flat and by selecting words for v_1 and v_2, we can get words for the other variables using σ. Furthermore, for an arbitrary selection of words from the languages of v_1 and v_2, the corresponding string assignment is a model of the string constraint. For instance, $\nu = \{x \mapsto a, w \mapsto aa, y \mapsto b, z \mapsto ab, v_1 \mapsto a, v_2 \mapsto b\}$ is a model for a selection of words $v_1 \mapsto a \in \mathsf{Lang}(v_1)$ and $v_2 \mapsto b \in \mathsf{Lang}(v_2)$. Therefore, (Lang, σ) is a stable solution. ⌟

Noodlification. A key notion of the stabilization-based procedure introduced in [14] and later extended to string constraints with lengths [18] is *noodlification*. Noodlification allows to steer the generation of possible equation splits (or alignments) only to feasible ones. An *equation split* is a splitting of variables to a concatenation of new fresh variables so that boundaries between variables on both sides of an equation match. A variable split is performed together with splitting the variable's language in Lang. A feasible split has a property that the fresh variables have nonempty languages. For instance, for the equation $xyu = wz$ with $\mathsf{Lang} = \{x \mapsto a^*, p \mapsto (a + b)^* \mid p \in \{y, u, w, z\}\}$, one possible feasible split is $\{x = v_1v_2, y = v_3, u = v_4, w = v_1, z = v_2v_3v_4\}$ together with the language assignment $\mathsf{Lang}' = \mathsf{Lang} \cup \{v_1 \mapsto a^*, v_2 \mapsto a^*, v_3 \mapsto (a + b)^*, v_4 \mapsto (a + b)^*\}$.

The length-aware noodlification, denoted as $\mathsf{noodlify}(t = s, \mathsf{Lang})$, generates a set of all feasible splits of the equation $t = s$. A split is a pair (E, Lang') where E is a set of new equations and Lang' is a language assignment of fresh variables. An efficient implementation of noodlification uses nondeterministic finite automata (NFAs) to represent the language assignment[3]. The noodlification then constructs ϵ-preserving products $\mathcal{P}$ of ϵ-concatenated automata of t and s. Each side has a different ϵ-symbol serving as a delimiter between variables of each equation side. The resulting feasible splits are then generated by inspecting parts of the product automaton $\mathcal{P}$.

Simplified stabilization-based procedure. In this paragraph we describe a simplified length-aware stabilization-based procedure. We briefly discuss particular optimizations of the procedure introduced in [14, 18] at the end of the paragraph. The stabilization-based procedure starts from the tuple $(\mathcal{E}, \mathsf{Lang}_{\mathcal{R}}, \mathsf{id}_{\mathbb{X}})$ where $\mathcal{E}$ are string equations and $\mathsf{Lang}_{\mathcal{R}}$ is the language assignment obtained from the initial regular constraint $\mathcal{R}$. It then proceeds by applying inference rules given below, trying to reach a tuple $(\emptyset, \mathsf{Lang}, \sigma)$ such that (Lang, σ) is a stable solution (which denotes satisfiability of the original constraint) or a tuple where $\mathsf{Lang}(x) = \emptyset$ for some $x \in \mathbb{X}$ (which denotes unsatisfiability).

The first rule is Align&Split, which performs noodlification on an equation $s = t$ and adds equations representing generated feasible alignments, removing $s = t$ from the set of equations (the formula on the right-hand side of a rule denotes a condition when the rule can be applied):

[3] We start with NFAs obtained from initial regular constraints and in each step perform only regularity-preserving operations.

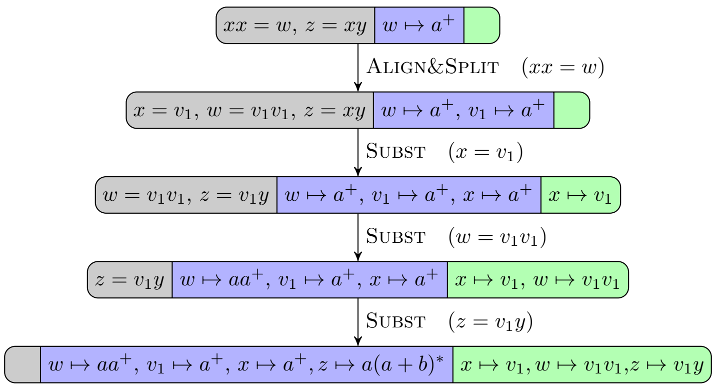

$xx = w,\ z = xy$ | $w \mapsto a^+$
Align&Split $(xx = w)$
$x = v_1,\ w = v_1v_1,\ z = xy$ | $w \mapsto a^+,\ v_1 \mapsto a^+$
Subst $(x = v_1)$
$w = v_1v_1,\ z = v_1y$ | $w \mapsto a^+,\ v_1 \mapsto a^+,\ x \mapsto a^+$ | $x \mapsto v_1$
Subst $(w = v_1v_1)$
$z = v_1y$ | $w \mapsto aa^+,\ v_1 \mapsto a^+,\ x \mapsto a^+$ | $x \mapsto v_1,\ w \mapsto v_1v_1$
Subst $(z = v_1y)$
$w \mapsto aa^+,\ v_1 \mapsto a^+,\ x \mapsto a^+, z \mapsto a(a+b)^*$ | $x \mapsto v_1, w \mapsto v_1v_1, z \mapsto v_1y$

Figure 1 Part of a proof graph generated by rules of simplified stabilization-based procedure for the string constraint $xy = z \land w = xx \land w \in a^+$ over the alphabet $\{a, b\}$. Tuples $(\mathcal{E}, \mathsf{Lang}, \sigma)$ are visualized by nodes $\mathcal{E}$ Lang σ (we omit there implicit language assignments of variables to $(a+b)^*$ as well as identity substitutions $x \mapsto x$). Edges are labelled by the used rule together with the selected equation. The language assignment and substitution in the bottom node form a stable solution.

$$\textsc{Align\&Split}\colon \frac{(\ \mathcal{E} \uplus \{s = t\},\quad \mathsf{Lang},\quad \sigma\ )}{\{(\ \mathcal{E} \cup \mathit{Align}_i,\quad \mathsf{Lang}_i,\quad \sigma\ )\}_{i=1}^{n}} \quad \mathsf{noodlify}(s = t, \mathsf{Lang}) = \{(\mathit{Align}_i, \mathsf{Lang}_i)\}_{i=1}^{n}$$

The second rule, Subst, removes a simple equation $x = t$ where $x \in \mathbb{X}$ from the set of equations and transforms it into a substitution, which is then applied to the set of equations.

$$\textsc{Subst}\colon \frac{(\quad \mathcal{E} \uplus \{x = t\},\quad \mathsf{Lang},\quad \sigma,\quad)}{\{(\ \sigma_i'(\mathcal{E} \cup \mathit{Align}),\quad \mathsf{Lang}_i,\quad \sigma_i' \circ \sigma\ )\}_{i=1}^{n}} \ \varphi_{\textsc{Subst}}$$

where $\varphi_{\textsc{Subst}} \overset{\text{def}}{\Leftrightarrow} \mathsf{noodlify}(x = t, \mathsf{Lang}) = \{(\mathit{Align} \uplus \{x = t_i'\}, \mathsf{Lang}_i)\}_{i=1}^{n} \land \bigwedge_{i=1}^{n} \sigma_i' = \{x \mapsto t_i'\}$.

In [14, 18] these basic rules were extended by organizing equations in inclusion graphs, which then determine the order of selecting equations in Align&Split. Moreover, the notion of equations is replaced by inclusions allowing to distinguish sides of equations, which is important for the completeness on the *chain-free fragment* of string constraints [18, 6]. The extended rules also take into account information about length variables, which are propagated through alignments. During the concatenation of automata corresponding to non-length variables, ϵ symbols may be removed, leading to a significant reduction of the number of splits. Additional reduction of the number of generated splits during noodlification is achieved by eager simulation-based reduction applied on NFAs in the language assignment.

Generating length formulae. In order to check that a stable solution (Lang, σ) is satisfiable w.r.t. the initial length constraint $\mathcal{L}$, we first generate for each $x \in \mathbb{X}$ a LIA formula $\mathsf{len}(|x|, \mathsf{Lang}(x))$ describing all possible lengths of words of $\mathsf{Lang}(x)$. This can be achieved, e.g., by using the lasso-automata construction [4]. Consequently, we extend the formula $\bigwedge_{x \in \mathbb{X}} \mathsf{len}(|x|, \mathsf{Lang}(x))$ with the equality $|x| = |x_1| + \cdots + |x_n|$ for each $(x \mapsto x_1 \cdots x_n) \in \sigma$ and $|x_1| + \cdots + |x_n| = |y_1| + \cdots + |y_m|$ for each equation $x_1 \cdots x_n = y_1 \cdots y_m$ from $\mathcal{E}$ in order to obtain the resulting LIA formula φ_{len}. The LIA formula combines length constraints induced by initial and substituted equations and lengths of the substituted variables. Finally, we check if the LIA formula $\mathcal{L} \land \varphi_{\mathsf{len}}$ is satisfiable. Since we work with a stable solution, a LIA model of this formula can be used to construct a string model satisfying the obtained LIA model and, hence, satisfying the formula $\mathcal{E} \land \mathcal{R} \land \mathcal{L}$.

▶ **Example 2.** An example of obtaining a stable solution using rules of simplified stabilization-based procedure applied on the string constraint $xy = z \wedge w = xx \wedge w \in a^+$ is shown in Figure 1. The relevant part of the LIA formula φ_{len} corresponding to this stable solution is then given as

$$\begin{aligned}\varphi_{\mathsf{len}} \overset{\text{def}}{\Leftrightarrow} & |y| \geq 0 \wedge |x| \geq 1 \wedge |z| \geq 1 \wedge |w| \geq 2 \wedge |v_1| \geq 1 \wedge \\ & |x| + |x| = |w| \wedge |z| = |x| + |y| \wedge |x| = |v_1| \wedge |w| = |v_1| + |v_1| \wedge |z| = |v_1| + |y|. \quad \lrcorner\end{aligned}$$

5 String-Integer Conversions

We propose an extension of the stabilization-based procedure from Section 4 so that it can handle string constraints of the form $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$. We know that $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L}$ is satisfiable if there is a stable solution (Lang, σ) such that the LIA formula $\mathcal{L} \wedge \varphi_{\mathsf{len}}$ is satisfiable. From a high-level point of view, given a stable solution, our procedure creates a LIA formula $\varphi_{\mathsf{conv}} \overset{\text{def}}{\Leftrightarrow} \bigwedge_{c \in \mathcal{C}} \varphi_c$ encoding conversions from $\mathcal{C}$ so that the string constraint $\mathcal{E} \wedge \mathcal{R} \wedge \mathcal{L} \wedge \mathcal{C}$ is satisfiable iff the LIA formula $\mathcal{L} \wedge \varphi_{\mathsf{len}} \wedge \varphi_{\mathsf{conv}}$ is satisfiable.

Additional constraint generation. In order to simplify the resulting LIA formula for conversions φ_{conv}, we add additional membership constraints to the set of initial regular constraints $\mathcal{R}$. The stabilization-based procedure then works with this modified set of constraints. These additional constraints make the resulting formula smaller and since the stabilization-based procedure is particularly optimized for working with regular constraints, the constraints may significantly speed up the whole procedure. More specifically, we enrich regular constraints with formulae restricting the results of `from_int` and `from_code`; i.e., for each constraint $x = \texttt{from\_int}(k)$ from $\mathcal{C}$ we add the constraint $x \in ((\texttt{'1'} + \cdots + \texttt{'9'})(\texttt{'0'} + \cdots + \texttt{'9'})^*) + \epsilon$, restricting x to a valid representation of a number without leading zeros or ϵ, and for each $x = \texttt{from\_code}(k)$ from $\mathcal{C}$ we add the constraint $x \in \Sigma + \epsilon$ restricting x to be either the symbol whose code point we are computing or ϵ for invalid inputs.

5.1 Handling `to_int`

Let (Lang, σ) be a stable solution of a string constraint without conversions and $k = \texttt{to\_int}(x) \in \mathcal{C}$ be a conversion that we want to encode into a LIA formula $\varphi_{k=\texttt{to\_int}(x)}$. Generally speaking, generating a LIA formula from a regular language $\mathsf{Lang}(x)$ that represents all encoded numbers is not possible, because some non-linear function such as exponentiation is necessary [15]. For example, given the language $\{\texttt{'5'}\}\{\texttt{'0'}\}^*$, the corresponding formula in nonlinear arithmetic would be $\ell = 5 \cdot z \wedge \exists n \colon n \geq 0 \wedge z = 10^n$. For this reason, we assume that the language $\mathsf{Lang}(x)$ is *finite*. This restriction is relatively strong, but in combination with our underapproximations from Section 6, it appears to be permissive enough.

As $\mathsf{Lang}(x)$ is finite, we could easily enumerate all words of $\mathsf{Lang}(x)$ and encode into $\varphi_{k=\texttt{to\_int}(x)}$ that $\texttt{to\_int}(x)$ must be equal to one of these values. This would result in the formula $\bigvee_{w \in \mathsf{Lang}(x)} \texttt{to\_int}(x) = \texttt{to\_int}(w)$, with $\texttt{to\_int}(w)$ being the integer value that the string w represents (or -1 if w is a string that does not encode a number) as defined in Section 3. However, the number of possible words could easily blow up and the resulting formula would be too large. For example, for the language $\{\texttt{'0'}, \ldots, \texttt{'9'}\}^9$, we would need 10^9 disjuncts. We need a more succinct encoding. Moreover, the encoding must allow efficient handling the following two issues:

(i) We need to keep the correspondence between the length of x and the value of $\texttt{to\_int}(x)$. The string constraint may, for instance, look like $|x| \leq 3 \wedge z \geq 1000 \wedge z = \texttt{to\_int}(x)$, which is unsatisfiable due to the relation between the length of the string and the value the string represents.

(ii) There can be other variables whose conversion result depend on x. As an example, assume that $\sigma(x) = z_1 z_2$ and we also have to handle the conversion $m = \texttt{to\_int}(y)$ with $\sigma(y) = z_2 z_3$. This means that x must end with the same string that y starts with. The two formulae $\varphi_{k=\texttt{to\_int}(x)}$ and $\varphi_{m=\texttt{to\_int}(y)}$ obtained from the naive approach, however, do not encode this relation.

Succinct and efficient encoding. To achieve succinctness, we reduce the number of disjuncts by working with sets of strings that represent continuous intervals of numbers. For example, if $\mathsf{Lang}(x) = \{\texttt{'0'}, \ldots, \texttt{'3'}, \texttt{'5'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}$, then we get two intervals of numbers $[0 - 399]$ and $[500 - 599]$, and we can create a significantly smaller LIA formula $\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_int}(x) \wedge (0 \leq \texttt{to\_int}(x) \leq 399 \vee 500 \leq \texttt{to\_int}(x) \leq 599)$. The two issues discussed above are then handled as follows:

(i) We relate each interval with the corresponding length.

(ii) We work on the level of substituted variables (according to σ) instead of the original ones. Because σ is flat, we can directly give a formula to define $\texttt{to\_int}$ values of the substituted variables and then use these to define the original ones. Regarding the problematic example above, because we know that σ is flat, it holds that $\sigma(z_1) = z_1$, $\sigma(z_2) = z_2$, and $\sigma(z_3) = z_3$, so we construct formulae for $\texttt{to\_int}(z_1)$, $\texttt{to\_int}(z_2)$, and $\texttt{to\_int}(z_3)$, and then combine them to get the value of the original $\texttt{to\_int}(x)$ and $\texttt{to\_int}(y)$.

Encoding $\texttt{to\_int}(x_i)$. To encode $\texttt{to\_int}(x)$, assuming that $\sigma(x) = x_1 \cdots x_n$, we need to first encode $\texttt{to\_int}(x_i)$ for each x_i. We split this encoding into two formulae: $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)}$, which encodes all words from $\mathsf{Lang}^{\mathsf{digit}}_{x_i} = \mathsf{Lang}(x_i) \cap \{\texttt{'0'}, \ldots, \texttt{'9'}\}^*$, i.e., words containing only digits; and $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)}$, which encodes all words from $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i} = \mathsf{Lang}(x_i) \setminus \{\texttt{'0'}, \ldots, \texttt{'9'}\}^*$, i.e., words containing at least one non-digit symbol.

To define $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)}$, we introduce the following notation. Let $L_{x_i} = \{|w| \mid w \in \mathsf{Lang}^{\mathsf{digit}}_{x_i}\}$ denote the set of all possible lengths of words containing only digits in $\mathsf{Lang}(x_i)$. Because we assume that $\mathsf{Lang}(x_i)$ is finite, then also L_{x_i} will be finite. Furthermore, let $I_{x_i}(m)$, for $m \in L_{x_i}$, be the set of pairs $(\mathit{low}, \mathit{high})$ of words of the length m from $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ that encode the largest possible continuous intervals of numbers of the form $(\texttt{to\_int}(\mathit{low}), \texttt{to\_int}(\mathit{high}))$. For example, for $\mathsf{Lang}(x_i) = \{\epsilon, \texttt{'0'}, \texttt{'1'}\}\{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}\}\{\texttt{'0'}, \ldots, \texttt{'9'}\}$ we have $L_{x_i} = \{2, 3\}$, $I_{x_i}(2) = \{(\texttt{'00'}, \texttt{'29'})\}$, and $I_{x_i}(3) = \{(\texttt{'000'}, \texttt{'029'}), (\texttt{'100'}, \texttt{'129'})\}$. Formally, given an interval encoding $(\mathit{low}, \mathit{high}) \in I_{x_i}(m)$, we have that $|\mathit{low}| = |\mathit{high}| = m$ and $\texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(\mathit{high})$. Further, the words with length m that encode numbers $\texttt{to\_int}(\mathit{low}) - 1$ and $\texttt{to\_int}(\mathit{high}) + 1$ are not in $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ and all words w whose length is m and $\texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(w) \leq \texttt{to\_int}(\mathit{high})$ must belong to $\mathsf{Lang}(x_i)$. The formula is then defined as

$$\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\ell \in L_{x_i}} \left(|x_i| = \ell \wedge \bigvee_{(\mathit{low}, \mathit{high}) \in I_{x_i}(\ell)} \texttt{to\_int}(\mathit{low}) \leq \texttt{to\_int}(x_i) \leq \texttt{to\_int}(\mathit{high}) \right)$$

where $\texttt{to\_int}(\mathit{low})$ and $\texttt{to\_int}(\mathit{high})$ denote the integers represented by the string literals *low* and *high* respectively. The formula says that for each length ℓ, $\texttt{to\_int}(x_i)$ must be a number encoded by one of the words from the language $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ whose length is ℓ. Let us note that for $\ell = 0$ there is only one possible interval (ϵ, ϵ) and for this case it holds that $\texttt{to\_int}(x_i) = -1$ (because of the fact that $\texttt{to\_int}(\epsilon) = -1$).

On the other hand, defining formula for words from $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i}$ is simple because for each such word w, we know that $\texttt{to\_int}(w) = -1$. Therefore, we can define it as

$$\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)} \overset{\text{def}}{\Leftrightarrow} \mathsf{len}(|x_i|, \mathsf{Lang}^{\mathsf{nondigit}}_{x_i}) \wedge \texttt{to\_int}(x_i) = -1$$

where $\mathsf{len}(|x_i|, \mathsf{Lang}^{\mathsf{nondigit}}_{x_i})$, defined in Section 4, is a LIA formula encoding that $|x_i|$ must be the length of some word from the language $\mathsf{Lang}^{\mathsf{nondigit}}_{x_i}$. Note that because this formula can be created even for an infinite language, we can relax the condition on finiteness of the language $\mathsf{Lang}(x_i)$ to only finiteness of the $\mathsf{Lang}^{\mathsf{digit}}_{x_i}$ part.

Encoding $\texttt{to\_int}(x)$. Having defined the encodings of $\texttt{to\_int}(x_i)$ for all x_i in $\sigma(x) = x_1 \cdots x_n$, we can now use them to define $\varphi_{k=\texttt{to\_int}(x)}$. Again, we split the definition into two formulae, but instead of splitting it based on whether the given word contains a digit, we split it into two cases, based on whether x represents a valid number or not, into formulae $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ and $\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}$ respectively.

We start with the definition of a LIA formula $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ expressing possible valuations of inputs that represent valid numbers. Notice that for a given combination $(\ell_1, \ldots, \ell_n) \in L_{x_1} \times \cdots \times L_{x_n}$ of lengths, we can easily compute the value $\texttt{to\_int}(x)$ by summing the values of $\texttt{to\_int}(x_i)$ multiplied by the correct power of ten. Therefore, we define $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ as the following formula:

$$\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\substack{\ell_1 \in L_{x_1} \\ \vdots \\ \ell_n \in L_{x_n} \\ \ell_1 + \cdots + \ell_n \neq 0}} \left(\texttt{to\_int}(x) = \sum_{1 \le i \le n} \left(\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n} \cdot \operatorname{sgn} \ell_i \right) \wedge \psi[l_1 \ldots l_n] \right)$$

where sgn is the *sign function* (that sends ℓ_i to 1 if it is positive and to 0 if it is itself 0), the corresponding summand $\texttt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n}$ has to be equal to 0, and

$$\psi[l_1 \ldots l_n] \overset{\text{def}}{\Leftrightarrow} \bigwedge_{1 \le i \le n} \left(|x_i| = \ell_i \wedge (\texttt{to\_int}(x_i) = -1 \Rightarrow \ell_i = 0) \right)$$

connects the length $|x_i|$ with the value of ℓ_i and discards values of x_i that do not represent numbers (except for the empty word ϵ). Note that if some (but not all) variable x_i is an empty word, then $x_1 \cdots x_n$ still represents a valid number. Furthermore, because $|x| = |x_1| + \cdots + |x_n|$ is a part of φ_{len}, the connection between $|x|$ and the value of $\texttt{to\_int}(x)$ can be kept by putting restrictions only on the values $|x_1|, \ldots, |x_n|$.

We now define formula $\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)}$ encoding the situation when x does not represent a number and $\texttt{to\_int}(x)$ should be equal to -1:

$$\varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} \texttt{to\_int}(x) = -1 \wedge \left(|x| = 0 \vee \bigvee_{1 \le i \le n} \left(\texttt{to\_int}(x_i) = -1 \wedge |x_i| \neq 0 \right) \right)$$

The variable x does not represent a number if it is ϵ or if some x_i does not represent a number. However, x_i might not represent a number even in the case that $x_i = \epsilon$, but as already mentioned, x can still be a valid representation of a number, so we discard these cases.

Finally, the resulting formula $\varphi_{k=\texttt{to\_int}(x)}$ is given as

$$\varphi_{k=\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_int}(x) \wedge \left(\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \vee \varphi^{\mathsf{invalid}}_{\texttt{to\_int}(x)} \right) \wedge \bigwedge_{1 \le i \le n} \left(\varphi^{\mathsf{digit}}_{\texttt{to\_int}(x_i)} \vee \varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)} \right).$$

We note that in the case $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ is true, its conjunct $\psi[l_1 \ldots l_n]$ forces $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(x_i)}$ to be false for all x_i.

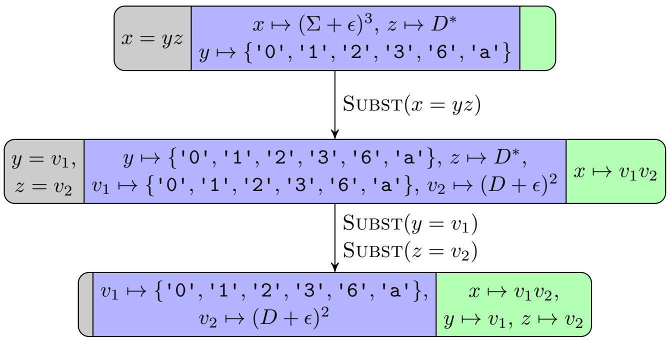

$x = yz$
$x \mapsto (\Sigma+\epsilon)^3$, $z \mapsto D^*$
$y \mapsto \{$'0', '1', '2', '3', '6', 'a'$\}$
SUBST$(x = yz)$
$y = v_1$, $z = v_2$
$y \mapsto \{$'0', '1', '2', '3', '6', 'a'$\}$, $z \mapsto D^*$,
$v_1 \mapsto \{$'0', '1', '2', '3', '6', 'a'$\}$, $v_2 \mapsto (D+\epsilon)^2$
$x \mapsto v_1v_2$
SUBST$(y = v_1)$
SUBST$(z = v_2)$
$v_1 \mapsto \{$'0', '1', '2', '3', '6', 'a'$\}$,
$v_2 \mapsto (D+\epsilon)^2$
$x \mapsto v_1v_2$,
$y \mapsto v_1$, $z \mapsto v_2$

Figure 2 A part of the proof graph generated by the rules of the simplified stabilization procedure for the string constraint $x = yz \land x \in (\Sigma+\epsilon)^3 \land y \in \{\texttt{'0'},\texttt{'1'},\texttt{'2'},\texttt{'3'},\texttt{'6'},\texttt{'a'}\} \land z \in D^*$, where $D = \{\texttt{'0'},\ldots,\texttt{'9'}\}$.

▶ **Example 3.** Let $D = \{\texttt{'0'},\ldots,\texttt{'9'}\}$ be the set of digits and

$$\psi \overset{\text{def}}{\Leftrightarrow} x = yz \land x \in (\Sigma+\epsilon)^3 \land y \in \{\texttt{'0'},\texttt{'1'},\texttt{'2'},\texttt{'3'},\texttt{'6'},\texttt{'a'}\} \land z \in D^* \land i = \texttt{to\_int}(x)$$

a string constraint that we want to solve. Figure 2 shows a run of the stabilization procedure on this string constraint (without the to_int conversion) that results in the stable solution (Lang,σ), where $\mathsf{Lang}(v_1) = \{\texttt{'0'},\texttt{'1'},\texttt{'2'},\texttt{'3'},\texttt{'6'},\texttt{'a'}\}$, $\mathsf{Lang}(v_2) = (D+\epsilon)^2$, $\sigma(x) = v_1v_2$, $\sigma(y) = v_1$, and $\sigma(z) = v_2$ (mappings of other variables in Lang and σ are not relevant). The relevant part of the LIA formula φ_{len} corresponding to this stable solution is then given as

$$\varphi_{\mathsf{len}} \overset{\text{def}}{\Leftrightarrow} (|v_1| = 1 \lor |v_1| = 2) \land 0 \le |v_2| \le 2 \land |x| = |v_1| + |v_2| \land |y| = |v_1| \land |z| = |v_2|.$$

We need to create the formula $\varphi_{i=\texttt{to\_int}(x)}$ and because $\sigma(x) = v_1v_2$, we need to first create the formulae $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_1)}$, $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_1)}$, $\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)}$, and $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_2)}$. Starting with v_1, we have $\mathsf{Lang}^{\mathsf{digit}}_{v_1} = \{\texttt{'0'},\texttt{'1'},\texttt{'2'},\texttt{'3'},\texttt{'6'}\}$, $\mathsf{Lang}^{\mathsf{nondigit}}_{v_1} = \{\texttt{'a'}\}$, and $L_{v_1} = \{1\}$. Therefore, we get

$$\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_1)} \overset{\text{def}}{\Leftrightarrow} |v_1| = 1 \land (0 \le \texttt{to\_int}(v_1) \le 3 \lor \texttt{to\_int}(v_1) = 6) \text{ and}$$
$$\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_1)} \overset{\text{def}}{\Leftrightarrow} |v_1| = 1 \land \texttt{to\_int}(v_1) = -1.$$

For v_2, we have $\mathsf{Lang}^{\mathsf{digit}}_{v_2} = (D+\epsilon)^2$, $\mathsf{Lang}^{\mathsf{nondigit}}_{v_2} = \emptyset$, and $L_{v_2} = \{0,1,2\}$. We obtain

$$\begin{aligned}\varphi^{\mathsf{digit}}_{\texttt{to\_int}(v_2)} \overset{\text{def}}{\Leftrightarrow} (|v_2| = 0 \land \texttt{to\_int}(v_2) = -1) \lor (|v_2| = 1 \land 0 \le \texttt{to\_int}(v_2) \le 9)\\ \lor (|v_2| = 2 \land 0 \le \texttt{to\_int}(v_2) \le 99)\end{aligned}$$

and, since $\mathsf{Lang}^{\mathsf{nondigit}}_{v_2}$ is empty, $\varphi^{\mathsf{nondigit}}_{\texttt{to\_int}(v_2)}$ is false. The formula $\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)}$ is then given as

$$\begin{aligned}\varphi^{\mathsf{valid}}_{\texttt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} & \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \land |v_1| = 1 \land |v_2| = 0 \land \texttt{to\_int}(v_1) \ne -1\big)\\ & \lor \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \cdot 10 + \texttt{to\_int}(v_2) \land |v_1| = 1\\ & \qquad \land |v_2| = 1 \land \texttt{to\_int}(v_1) \ne -1 \land \texttt{to\_int}(v_2) \ne -1\big)\\ & \lor \big(\texttt{to\_int}(x) = \texttt{to\_int}(v_1) \cdot 100 + \texttt{to\_int}(v_2) \land |v_1| = 1\\ & \qquad \land |v_2| = 2 \land \texttt{to\_int}(v_1) \ne -1 \land \texttt{to\_int}(v_2) \ne -1\big)\end{aligned}$$

and $\varphi^{\mathsf{invalid}}_{\mathtt{to\_int}(x)}$ as

$$\begin{aligned}\varphi^{\mathsf{invalid}}_{\mathtt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} {}& \mathtt{to\_int}(x) = -1 \wedge \\ & \big(|x| = 0 \vee (\mathtt{to\_int}(v_1) = -1 \wedge |v_1| \neq 0) \vee (\mathtt{to\_int}(v_2) = -1 \wedge |v_2| \neq 0)\big).\end{aligned}$$

The final formula $\varphi_{i=\mathtt{to\_int}(x)}$ is then

$$\varphi_{i=\mathtt{to\_int}(x)} \overset{\text{def}}{\Leftrightarrow} i = \mathtt{to\_int}(x) \wedge \left(\varphi^{\mathsf{valid}}_{\mathtt{to\_int}(x)} \vee \varphi^{\mathsf{invalid}}_{\mathtt{to\_int}(x)}\right) \wedge \left(\varphi^{\mathsf{digit}}_{\mathtt{to\_int}(v_1)} \vee \varphi^{\mathsf{nondigit}}_{\mathtt{to\_int}(v_1)}\right) \wedge \varphi^{\mathsf{digit}}_{\mathtt{to\_int}(v_2)}$$

which is the only conjunct of φ_{conv}. ⌟

5.2 Handling `from_int`

We briefly describe the handling of `from_int`. Assume that $x = \mathtt{from\_int}(k)$ is the conversion from $\mathcal{C}$ that we want to encode and $\sigma(x) = x_1 \cdots x_n$. We proceed similarly to the `to_int` case, but instead of encoding the possible values of the result x, we will instead restrict the argument k to values that yield the given possible values of the result x. The formula $\varphi^{\mathsf{valid}}_{x=\mathtt{from\_int}(k)}$ expressing valid conversions is the same as $\varphi^{\mathsf{valid}}_{\mathtt{to\_int}(x)}$ after replacing $\mathtt{to\_int}(x)$ with k:

$$\varphi^{\mathsf{valid}}_{x=\mathtt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} \bigvee_{\substack{\ell_1 \in L_{x_1} \\ \vdots \\ \ell_n \in L_{x_n} \\ \ell_1 + \cdots + \ell_n \neq 0}} \left(k = \sum_{1 \leq i \leq n} \left(\mathtt{to\_int}(x_i) \cdot 10^{\ell_{i+1} + \cdots + \ell_n} \cdot \operatorname{sgn} \ell_i\right) \wedge \psi[l_1 \ldots l_n]\right).$$

Furthermore, $\mathtt{from\_int}(k)$ always returns the string encoding a *non-negative* number k, except for the case when $k < 0$. In this case, it returns ϵ, therefore, the formula $\varphi^{\mathsf{invalid}}_{x=\mathtt{from\_int}(k)}$ is defined as

$$\varphi^{\mathsf{invalid}}_{x=\mathtt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} k < 0 \wedge |x| = 0.$$

The resulting formula $\varphi_{x=\mathtt{from\_int}(k)}$ is then given as

$$\varphi_{x=\mathtt{from\_int}(k)} \overset{\text{def}}{\Leftrightarrow} \left(\varphi^{\mathsf{valid}}_{x=\mathtt{from\_int}(k)} \vee \varphi^{\mathsf{invalid}}_{x=\mathtt{from\_int}(k)}\right) \wedge \bigwedge_{1 \leq i \leq n} \left(\varphi^{\mathsf{digit}}_{\mathtt{to\_int}(x_i)} \vee \varphi^{\mathsf{nondigit}}_{\mathtt{to\_int}(x_i)}\right).$$

▶ **Example 4.** Let us take the string constraint from Example 3 extended with $i \leq j \wedge z = \mathtt{from\_int}(j)$. As we added only LIA and conversion constraints, the stable solution (Lang, σ) from Example 3 stays the same. We know that $\sigma(z) = v_2$, $\mathsf{Lang}(v_2) = (D + \epsilon)^2$, and $L_{v_2} = \{0, 1, 2\}$. We also already have $\varphi^{\mathsf{digit}}_{\mathtt{to\_int}(v_2)}$ from Example 3 ($\varphi^{\mathsf{nondigit}}_{\mathtt{to\_int}(v_2)}$ is unsatisfiable), therefore, in order to encode $\varphi_{z=\mathtt{from\_int}(j)}$, we only need

$$\begin{aligned}\varphi^{\mathsf{valid}}_{z=\mathtt{from\_int}(j)} &\overset{\text{def}}{\Leftrightarrow} (j = \mathtt{to\_int}(v_2) \wedge |v_2| = 1 \wedge \mathtt{to\_int}(v_2) \neq -1) \vee \\ &\quad\;\; (j = \mathtt{to\_int}(v_2) \wedge |v_2| = 2 \wedge \mathtt{to\_int}(v_2) \neq -1) \text{ and} \\ \varphi^{\mathsf{invalid}}_{z=\mathtt{from\_int}(j)} &\overset{\text{def}}{\Leftrightarrow} j < 0 \wedge |z| = 0.\end{aligned}$$

We can then add

$$\varphi_{z=\mathtt{from\_int}(j)} \overset{\text{def}}{\Leftrightarrow} \left(\varphi^{\mathsf{valid}}_{z=\mathtt{from\_int}(i)} \vee \varphi^{\mathsf{invalid}}_{z=\mathtt{from\_int}(i)}\right) \wedge \varphi^{\mathsf{digit}}_{\mathtt{to\_int}(v_2)}$$

to φ_{conv}. ⌟

5.3 Handling `to_code`

The semantic of $\texttt{to\_code}(x)$ is simple (see Section 3): if $|x| = 1$, then $\texttt{to\_code}(x)$ represents the code point of the symbol x, otherwise the result is -1. We want to encode this behavior in the formula $\varphi_{k=\texttt{to\_code}(x)}$. Assuming that $\sigma(x) = x_1 \cdots x_n$, we will first, for each x_i, encode the value $\texttt{to\_code}(x_i)$ in a formula $\varphi_{\texttt{to\_code}(x_i)}$, so that we relate the substituted variables between multiple `to_code` conversions (similarly as in `to_int` case). We could take a naive approach and enumerate all possible symbols of x_i (there is only a finite number of possible symbols, so this method is complete). However, such a naive approach could easily blow-up.

In order to overcome this blow-up problem, we can notice that, despite the high number of possible symbols, the input string formula usually uses only a small subset of them (usually a subset of ASCII symbols). We therefore restrict the alphabet Σ only to these symbols, all digit symbols, and one special symbol δ that represents all *unused*[4] *non-digit* symbols (if there are any). Using only δ for all unused symbols has no impact on the stabilization-based procedure from Section 4, as it handles only a conjunction of positive equations with regular and length constraints, where unused symbols "behave in the same way". However, digit symbols are important for `to_int`/`from_int` conversions. We therefore keep all of them in Σ and δ then represents all unused symbols that are not digits.

We then encode the fact that either $\texttt{to\_code}(x_i)$ is a code point of the symbol x_i, or, if x_i is not a single symbol, $\texttt{to\_code}(x_i) = -1$:

$$\varphi_{\texttt{to\_code}(x_i)} \overset{\text{def}}{\Leftrightarrow} \Big(\big(|x_i| = 1 \land \bigvee_{a \in \mathsf{Lang}(x_i) \cap \Sigma} \psi(a) \big) \lor \big(|x_i| \neq 1 \land \texttt{to\_code}(x_i) = -1 \big) \Big)$$

where $\psi(a)$ encodes that $\texttt{to\_code}(x_i)$ is equal to the code-point of the symbol a as follows:

- For a normal symbol $a \neq \delta$, we have $\psi(a) \overset{\text{def}}{\Leftrightarrow} \texttt{to\_code}(x_i) = \texttt{to\_code}(a)$ (recall that $\texttt{to\_code}(a)$ for $a \in \Sigma \setminus \{\delta\}$ denotes the numeric value of the code point of the symbol a).
- For δ, we have to encode the fact that $\texttt{to\_code}(\delta)$ can acquire any possible code point of unused symbols. Therefore, given that $\mathsf{maxchar}$ denotes the largest possible code point[5],

$$\psi(\delta) \overset{\text{def}}{\Leftrightarrow} 0 \leq \texttt{to\_code}(x_i) \leq \mathsf{maxchar} \land \bigwedge_{b \in \mathsf{Lang}(x_i) \cap (\Sigma \setminus \{\delta\})} \texttt{to\_code}(x_i) \neq \texttt{to\_code}(b)$$

We can then define $\varphi_{\texttt{to\_code}(x)}$ as

$$\varphi_{\texttt{to\_code}(x)} \overset{\text{def}}{\Leftrightarrow} \Big(\big(|x| = 1 \land \texttt{to\_code}(x) \neq -1 \land \bigvee_{1 \leq i \leq n} \texttt{to\_code}(x) = \texttt{to\_code}(x_i) \big) \lor \big(|x| \neq 1 \land \texttt{to\_code}(x) = -1 \big) \Big)$$

relating the value x with the substituted variables x_i. The second part of the formula handles the case when x is not a symbol. For the case that x is a symbol, there must be exactly one x_i that is also the (same) symbol, and all other x_j, for $i \neq j$, must be empty strings. For each such x_j, we have $\texttt{to\_code}(x_j) = -1$, so by requiring $\texttt{to\_code}(x) \neq -1$, we force the equality $\texttt{to\_code}(x) = \texttt{to\_code}(x_i)$. The resulting formula is then

$$\varphi_{k=\texttt{to\_code}(x)} \overset{\text{def}}{\Leftrightarrow} k = \texttt{to\_code}(x) \land \varphi_{\texttt{to\_code}(x)} \land \bigwedge_{1 \leq i \leq n} \varphi_{\texttt{to\_code}(x_i)}.$$

[4] Unused here means that they are not *explicitly* used, so for example in the formula $x \in \Sigma \land y \in \{a, b\}$, the explicitly used symbols are only a and b, and δ will represent all other symbols. We then get a formula with three symbols: $x \in \{a, b, \delta\} \land y \in \{a, b\}$.

[5] According to the SMT standard [10], it is the number 196,607.

Connecting string and code conversions. Finally, we need to address the problem of relating $\texttt{to\_code}$ and $\texttt{to\_int}$. For example, we could have both conversions $\texttt{to\_int}(x)$ and $\texttt{to\_code}(y)$ with $\sigma(x) = z_1z_2$ and $\sigma(y) = z_2z_3$. Right now, $\texttt{to\_int}(z_2)$ and $\texttt{to\_code}(z_2)$ can have some valid values, which are, however, not related. For example, if $\mathsf{Lang}(z_2) = \{\texttt{'4'}, \texttt{'5'}\}$, then we could end up with the situation where $\texttt{to\_int}(z_2) = 5$ but $\texttt{to\_code}(z_2) = 52$, which is the code point of $\texttt{'4'}$.

Therefore, for each variable x for which both $\varphi_{\texttt{to\_int}(x)}$ and $\varphi_{\texttt{to\_code}(x)}$ are defined, we create the following formula

$$|x| \neq -1 \vee (\texttt{to\_int}(x) = -1 \wedge \neg(48 \leq \texttt{to\_code}(x) \leq 57)) \vee (\texttt{to\_code}(x) = \texttt{to\_int}(x)+48),$$

which we add to the conversion formula φ_{conv}. This formula represents three (non-necessarily disjoint) cases: (i) the string x is not a symbol, which means that $\texttt{to\_code}(x) = -1$ and $\texttt{to\_int}(x)$ does not depend on the value of $\texttt{to\_code}(x)$; (ii) x is not a digit, therefore $\texttt{to\_code}(x)$ cannot lie in the interval $[48, 57]$, which corresponds to digits $\texttt{'0'}$ through $\texttt{'9'}$; and (iii) x is a digit, so $\texttt{to\_code}(x)$ must be its code point.

▶ **Example 5.** We further extend the string constraint from Example 4 by the constraint $|x| = \texttt{to\_code}(y)$. Again, we only added a conversion to the formula, therefore the stable solution (Lang, σ) from Example 3 stays the same, and we only need to encode the formula $\varphi_{|x|=\texttt{to\_code}(y)}$ with $\sigma(y) = v_1$ and $\mathsf{Lang}(v_1) = \{\texttt{'0'}, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'6'}, \texttt{'a'}\}$. We therefore get the following formulae:

$$\begin{aligned}
\varphi_{\texttt{to\_code}(v_1)} \overset{\text{def}}{\Leftrightarrow} \big(|v_1| = 1 \wedge (&\texttt{to\_code}(v_1) = 48 \vee \\
&\texttt{to\_code}(v_1) = 49 \vee \\
&\texttt{to\_code}(v_1) = 50 \vee \\
&\texttt{to\_code}(v_1) = 51 \vee \\
&\texttt{to\_code}(v_1) = 54 \vee \\
&\texttt{to\_code}(v_1) = 97)\big) \vee \\
\big(|v_1| \neq 1 \wedge &\texttt{to\_code}(v_1) = -1\big) \\
\varphi_{\texttt{to\_code}(y)} \overset{\text{def}}{\Leftrightarrow} \big(|y| = 1 \wedge &\texttt{to\_code}(y) \neq -1 \wedge \texttt{to\_code}(y) = \texttt{to\_code}(v_1)\big) \vee \\
\big(|y| \neq 1 \wedge &\texttt{to\_code}(y) = -1\big) \\
\varphi_{|x|=\texttt{to\_code}(y)} \overset{\text{def}}{\Leftrightarrow} |x| = &\texttt{to\_code}(y) \wedge \varphi_{\texttt{to\_code}(y)} \wedge \varphi_{\texttt{to\_code}(x_i)}
\end{aligned}$$

Note that in Example 3 we defined $\texttt{to\_int}(v_1)$ and we have also now defined $\texttt{to\_code}(v_1)$, therefore we also need to add the formula connecting these two values together as explained in the previous paragraph. All in all, the final conversion formula φ_{conv} is then defined as

$$\begin{aligned}
\varphi_{\mathsf{conv}} \overset{\text{def}}{\Leftrightarrow} &\big(|v_1| \neq -1 \vee (\texttt{to\_int}(v_1) = -1 \wedge \neg(48 \leq \texttt{to\_code}(v_1) \leq 57)) \vee \\
&(\texttt{to\_code}(v_1) = \texttt{to\_int}(v_1) + 48)\big) \wedge \\
&\varphi_{i=\texttt{to\_int}(x)} \wedge \varphi_{z=\texttt{from\_int}(j)} \wedge \varphi_{|x|=\texttt{to\_code}(y)}.
\end{aligned}$$

⌟

5.4 Handling from_code

Similarly to $\texttt{from\_int}$, which is handled using $\texttt{to\_int}$, $\texttt{from\_code}$ is handled using $\texttt{to\_code}$. Given $x = \texttt{from\_code}(k)$ and $\sigma(x) = x_1 \cdots x_n$, the formula $\varphi_{x=\texttt{from\_code}(k)}$ is defined as

$$\begin{aligned}
\varphi_{x=\texttt{from\_code}(k)} \overset{\text{def}}{\Leftrightarrow} \bigg(&\Big(|x| = 1 \wedge k \neq -1 \wedge \bigvee_{1\leq i\leq n} k = \texttt{to\_code}(x_i)\Big) \vee \\
&\Big(|x| = 0 \wedge \neg(0 \leq k \leq \mathsf{maxchar})\Big)\bigg) \wedge \bigwedge_{1\leq i\leq n} \varphi_{\texttt{to\_code}(x_i)}
\end{aligned}$$

The only two differences from `to_code` are the following:

1. we are restricting the values of the argument k (instead of computing $\texttt{to\_code}(x)$) and
2. for the case that x is not a symbol (which must mean that $x = \epsilon$, as $\texttt{from\_code}(k)$ returns either a symbol for a valid code point or an empty string, hence the condition $|x| = 0$), the argument can be any integer that is an invalid code point.

5.5 Handling word disequations through `to_code`

The stabilization-based procedure from Section 4 assumes that $\mathcal{E}$ does not contain disequations. As shown in [18], every disequation $s \neq t$ can be encoded into a combination of equations and length constraints as

$$\varphi_{s\neq t} \overset{\text{def}}{\Leftrightarrow} |s| \neq |t| \vee \Big(s = x_1a_1y_1 \wedge t = x_2a_2y_2 \wedge |x_1| = |x_2| \wedge a_1 \in \Sigma \wedge a_2 \in \Sigma \wedge \overbrace{a_1 \neq a_2}^{\mathsf{dist}(a_1,a_2)}\Big)$$

where x_1, x_2, y_1, y_2, a_1, and a_2 are fresh variables and the disequation $a_1 \neq a_2$ between symbols can be encoded into a LIA formula $\mathsf{dist}(a_1, a_2)$ after the procedure returns a stable solution. Furthermore, if the original string constraint without disequations is chain-free, then it stays chain-free even if we add $\varphi_{s\neq t}$ to it. However, the transformation of $a_1 \neq a_2$ to LIA formula $\mathsf{dist}(a_1, a_2)$ from [18] is incompatible with the way we construct φ_{conv}. We can easily solve this by replacing $a_1 \neq a_2$ in $\varphi_{s\neq t}$ with $\texttt{to\_code}(a_1) \neq \texttt{to\_code}(a_2)$. Since our procedure can handle any `to_code` conversion, this extends one of the richest decidable fragments of string constraints, chain-free constraints with lengths and arbitrary disequations [18], by code-point conversions.

6 Implementation and Optimizations

We implemented the proposed technique into the string solver Z3-Noodler [34], which implements an optimized version of the stabilization-based procedure from Section 4 within the DPLL(T)-based SMT solver Z3. The LIA formula φ_{conv} for string-integer conversions is constructed after the procedure finds a stable solution. The formula φ_{conv} is appended to the length formula φ_{len} and the result is checked for satisfiability. On top of the proposed technique, we further introduced several optimizations, described below.

Underapproximation of `to_int`. We mentioned in Section 5 that the conversion $\texttt{to\_int}(x)$ expects the language $\mathsf{Lang}_x^{\mathsf{digit}}$, the subset of words from $\mathsf{Lang}(x)$ containing only digits, to be finite. Although this restriction usually holds in the benchmarks, we implemented an *underapproximation* to at least partially handle the case of infinite languages. In particular, if a variable occurring in a conversion has an infinite language after stabilization, we restrict the language to strings up to some fixed length. To be more concrete, if $\mathsf{Lang}_x^{\mathsf{digit}}$ of a variable x is an infinite language, we restrict it to $\mathsf{Lang}_x^{\mathsf{digit}} \cap (\Sigma + \epsilon)^m$ where m is the underapproximation parameter (we use $m = 5$ in the experiments). This approach is sound for the SAT case.

Supporting constraints. In order to keep the variable languages as precise as possible, we generate additional constraints (axioms) that are used to enrich the initial string constraint. One such kind of constraints is generated for conversions of the type $x = \texttt{from\_int}(k)$, which always result in x being assigned an infinite language (unless it is restricted by some other regular constraint). Therefore, we generate an additional constraint $k < 10^m \implies x \in (\Sigma + \epsilon)^m$ where m is the same underapproximation parameter as in the previous paragraph, which is then added to the initial constraint. Furthermore, we infer regular constraints from LIA formulae. For example, if we have $\texttt{to\_int}(x) = k$, with $k \in \mathbb{Z}$, we generate $x \in \texttt{'0'}^* \cdot \texttt{from\_int}(k)$. Moreover, if we have $|x| \leq k$, we can set $x \in (\Sigma + \epsilon)^k$.

Table 1 Results of experiments on all benchmark sets. For each benchmark set we give the number of solved instances, the number of unknowns, and the number of instances where the particular solver runs out of resources (timeout or out of memory). Moreover, we measure each value w.r.t. all instances in the benchmark set (*all*) as well as only to instances containing at least one conversion function (*conv*). The number of particular instances is given in the benchmark header.

	FullStrInt (16,968 | 16,130)						StringFuzz (11,618 | 1,608)						StrSmallRw (1,880 | 80)						Σ	
	solved		***unknown***		***OOR***		***solved***		***unknown***		***OOR***		***solved***		***unknown***		***OOR***		***solved***	
	all	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*	*all*	*conv*
Z3-Noodler	16,704	15,872	126	126	138	132	**11,616**	**1,606**	2	2	**0**	**0**	1,743	73	100	1	37	6	**30,063**	17,551
cvc5	**16,963**	**16,125**	**0**	**0**	5	5	10,915	1,579	**0**	**0**	703	29	**1,861**	**78**	2	2	**17**	**0**	29,739	**17,782**
Z3	16,729	15,896	**0**	**0**	239	234	11,081	1,565	**0**	**0**	537	43	1,821	**78**	**0**	**0**	59	2	29,631	17,539
OSTRICH	15,909	15,109	**0**	**0**	1,059	1,021	11,400	1,558	**0**	**0**	217	50	1,709	69	**0**	**0**	171	11	29,018	16,736
Z3-Noodler^{pr}	11,665	10,857	5,299	5,273	**4**	**0**	10,050	41	1,568	1,567	**0**	**0**	1,615	62	210	18	55	**0**	23,330	10,960

Interval computation. The solver Z3-Noodler internally represents each $\mathsf{Lang}(x)$ using NFAs. In order to efficiently create the set of intervals $I_{x_i}(n)$ for encoding $\mathtt{to\_int}(x_i)$ (cf. Section 5.1), we minimize the automaton for $\mathsf{Lang}(x_i)$, getting deterministic automaton with a finite language, i.e., from each state there is at most one transition per symbol and there are no loops. It is therefore easy to create intervals of digits going from each state, and then connect these short intervals into longer ones.

7 Experiments

Used tools and environment. We compared Z3-Noodler extended with the support of string-integer conversions (version 1.1.0) with the other state-of-the-art tools: cvc5 (version 1.1.2), Z3 (version 4.13.0), and OSTRICH (the latest commit `5dd2e10`). We also add comparison with the previous version 1.0.0 of Z3-Noodler (denoted as Z3-Noodler^{pr}). We do not include comparison with Z3-Trau and Z3str4 as they give incorrect results on some instances. The experiments were executed on a workstation with an AMD Ryzen 5 5600G CPU @ 3.8 GHz with 100 GiB of RAM running Ubuntu 22.04.4. The timeout was set to 120 s, memory limit was set to 8 GiB.

Benchmarks. For the experimental evaluation we selected all benchmark sets from SMT-LIB [9] (category `QF_SLIA`) containing string-integer conversions. Concretely, we took the FullStrInt (16,968 formulae), StringFuzz (11,618 formulae), and StrSmallRw (1,880 formulae) benchmark sets, which were also used in SMT-COMP'23. Except string-integer conversions, these benchmarks make heavy use of string (dis)equations and other string predicates and functions (e.g., `substr`, `indexof`, `at`, . . .) combined with length and regular constraints.

Results. The results summarizing the number of solved instances are given in Table 1 and the average running times are shown in Table 2. The scatter plots comparing the running times of Z3-Noodler and the other tools are given in Figure 3.

Regarding the FullStrInt benchmark set, Z3-Noodler has 138 timeouts (out of which 132 were on formulae with conversions) and 126 unknowns. The number of solved instances is comparable to Z3, but smaller compared to cvc5 (cvc5 solves all instances except 5). On the other hand, Z3-Noodler solved 795 instances more than OSTRICH. Concerning the running time, Z3-Noodler has the lowest average time followed by cvc5 (not counting Z3-Noodler^{pr} as it has almost 5,300 unknowns on FullStrInt). The inspection of timeouts of Z3-Noodler on FullStrInt revealed that the bottleneck on these examples is not the handling of string-integer conversions but handling of a complex combination of string predicates and functions (`substr`, `indexof`, . . .) in the stabilization-based procedure. We believe that these

Table 2 Average running times and standard deviation (in seconds) of *solved* instances.

	FullStrInt		StringFuzz		StrSmallRw	
	avg	*std*	*avg*	*std*	*avg*	*std*
Z3-Noodler	0.19	1.26	**0.03**	0.22	0.14	2.41
cvc5	0.30	1.42	2.81	12.55	**0.02**	0.56
Z3	1.21	6.90	4.14	15.39	0.12	2.86
OSTRICH	11.86	14.85	4.79	10.00	4.47	6.61
Z3-Noodler^{pr}	**0.02**	0.05	**0.03**	0.08	0.75	6.56

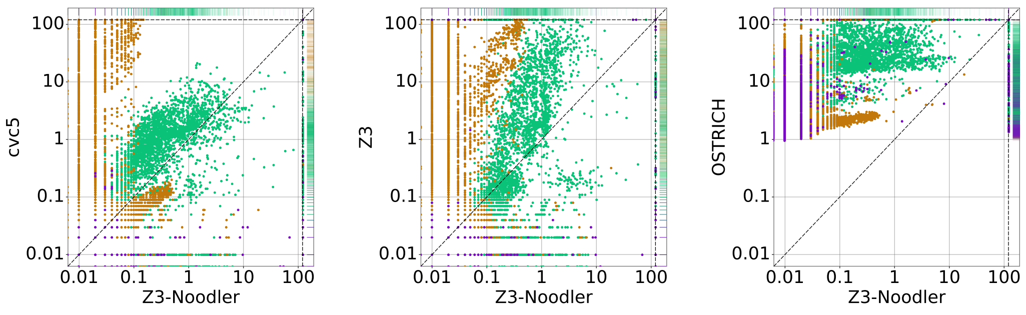
instances could be solved by a tailored preprocessing of input formulae and strengthening axioms for special cases of string functions and predicates. The unknowns of Z3-Noodler are caused by the inconclusive result in the case of underapproximation. It is evident from the table that Z3-Noodler can solve significantly more instances than Z3-Noodler^{pr}. Note that although Z3-Noodler^{pr} has no support for string-integer conversions, it can solve some instances, for example those containing unsatisfiable LIA parts or those whose satisfiability can be established without considering the string-integer conversions.

In the StringFuzz benchmark set, Z3-Noodler solved more instances than any other tool. In particular, Z3-Noodler has only 2 unknowns (caused by the underapproximation), which is significantly better than the runner-up OSTRICH. Concerning the running time on this benchmark set, Z3-Noodler has also the lowest average time.

The last considered benchmark set StrSmallRw is not very interesting. It contains only 80 formulae with conversions and from these, already 62 are solved by Z3-Noodler^{pr}, i.e., they can be solved without looking at conversions. If we take into account the whole benchmark, Z3-Noodler has a large amount of unknowns, which are caused by a limited support of the $\neg$`contains` predicate. The average running time of Z3-Noodler is comparable to Z3, but higher than the average time of cvc5, which is the fastest tool on this benchmark.

From the summary statistics in Table 1 it is evident that the proposed procedure and its integration with the stabilization-based procedure has a significant effect on the number of solved instances. The previous version Z3-Noodler^{pr} solved 6,734 instances less than Z3-Noodler. If we look at the number of solved cases from the perspective of all instances, Z3-Noodler solved the most formulae compared to any other tool. If we restrict our attention only to formulae with conversions, Z3-Noodler solves 231 instances less than cvc5, but more instances than Z3 and OSTRICH. From Figure 3 and the average running times in Table 2, it is obvious that Z3-Noodler can be (sometimes significantly) faster than other state-of-the-art tools pointing out to the efficiency of the proposed combined approach.

Discussion. The experimental evaluation shows that our proposed approach combining the stabilization-based procedure and the translation of the conversions into a LIA formula can (sometimes significantly) improve the scalability of solving complex string constraints containing string-integer conversions. Although the generated LIA formula might be large in the worst-case, based on our experiments, this worst case is avoided in real-world benchmarks (based on inspecting a couple of instances, the generated LIA formulae contained up to hundreds of simple atoms). The experiments also revealed that the highest impact on the LIA formula size was the interval separation (Section 5.1); without this feature the LIA formula often exploded. Furthermore, based on our experience, the generated LIA formula is easily solvable by Z3's LIA solver. The current bottleneck of the solver is not handling the string-integer conversions but handling the complex combination of string predicates and functions (as discussed at the results of FullStrInt).



(a) Z3-Noodler vs cvc5. **(b)** Z3-Noodler vs Z3. **(c)** Z3-Noodler vs OSTRICH.

Figure 3 Scatter plots comparing Z3-Noodler with cvc5, Z3, and OSTRICH. Times are in seconds, axes are logarithmic. The dashed lines represent unknowns and timeouts/out-of-memory. Colours distinguish benchmark sets: ● FullStrInt, ● StringFuzz, and ● StrSmallRw.

8 Conclusion

We have proposed an extension of the stabilization-based procedure with the handling of string-integer conversions. Based on a stable solution, the technique encodes the conversions into LIA formulae. We have proposed a series of optimizations in order to avoid a blow-up of the LIA formulae and implemented the approach in the tool Z3-Noodler. Our experimental evaluation on established benchmarks shows that our technique outperforms other state-of-the-art tools on many instances.

In the future, we plan to extend the precise handling of conversions from finite languages to languages with limited forms of iteration. Another direction could be a dynamic analysis of the underapproximation parameter and adjusting the parameter according to the input formula characteristics.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janku, Hsin-Hung Lin, Lukás Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 943–957. ACM, 2020. doi:10.1145/3385412.3386034.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Flatten and conquer: a framework for efficient analysis of string constraints. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 602–617. ACM, 2017. doi:10.1145/3062341.3062384.

3 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Lukás Holík, Ahmed Rezine, and Philipp Rümmer. Trau: SMT solver for string constraints. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–5. IEEE, 2018. doi:10.23919/FMCAD.2018.8602997.

4 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2014. doi:10.1007/978-3-319-08867-9_10.

5 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An SMT solver for string constraints. In *CAV'15*, volume 9206 of *LNCS*, pages 462–469. Springer, 2015.

6 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bui Phi Diep, Lukás Holík, and Petr Janků. Chain-free string constraints. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*, volume 11781 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2019. doi:10.1007/978-3-030-31784-3_16.

7 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. SolCMC: Solidity compiler's model checker. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 325–338. Springer, 2022. doi:10.1007/978-3-031-13185-1_16.

8 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

9 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

10 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB): Strings. https://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, 2023.

11 Murphy Berzish. *Z3str4: A Solver for Theories over Strings.* PhD thesis, University of Waterloo, Ontario, Canada, 2021. URL: https://hdl.handle.net/10012/17102.

12 Murphy Berzish, Joel D. Day, Vijay Ganesh, Mitja Kulczynski, Florin Manea, Federico Mora, and Dirk Nowotka. String theories involving regular membership predicates: From practice to theory and back. In Thierry Lecroq and Svetlana Puzynina, editors, *Combinatorics on Words*, pages 50–64, Cham, 2021. Springer International Publishing.

13 Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. An SMT solver for regular expressions and linear arithmetic over string length. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 289–312. Springer, 2021. doi:10.1007/978-3-030-81688-9_14.

14 František Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Word equations in synergy with regular constraints. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 403–423, Cham, 2023. Springer International Publishing.

15 Véronique Bruyàre, Georges Hansel, Christian Michaux, and Roger Villemaire. Logic and *p*-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society - Simon Stevin*, 1(2):191–238, 1994. doi:10.36045/bbms/1103408547.

16 Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 3(POPL):49:1–49:30, 2019. doi:10.1145/3290362.

17 Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Z3-Noodler: An automata-based string solver. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 24–33, Cham, 2024. Springer Nature Switzerland.

18 Yu-Fang Chen, David Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. Solving string constraints with lengths by stabilization. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. `doi:10.1145/3622872`.

19 Yu-Fang Chen, Vojtech Havlena, Ondrej Lengál, and Andrea Turrini. A symbolic algorithm for the case-split rule in string constraint solving. In Bruno C. d. S. Oliveira, editor, *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, volume 12470 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2020. `doi:10.1007/978-3-030-64437-6_18`.

20 Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. A symbolic algorithm for the case-split rule in solving word constraints with extensions. *Journal of Systems and Software*, 201:111673, 2023. `doi:10.1016/j.jss.2023.111673`.

21 Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. On solving word equations using SAT. In Emmanuel Filiot, Raphaël M. Jungers, and Igor Potapov, editors, *Reachability Problems - 13th International Conference, RP 2019, Brussels, Belgium, September 11-13, 2019, Proceedings*, volume 11674 of *Lecture Notes in Computer Science*, pages 93–106. Springer, 2019. `doi:10.1007/978-3-030-30806-3_8`.

22 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. `doi:10.1007/978-3-540-78800-3_24`.

23 Vijay Ganesh and Murphy Berzish. Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion, 2016. `arXiv:1605.09442`.

24 Quang Loc Le and Mengda He. A decision procedure for string logic with quadratic equations, regular expressions and length constraints. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 350–372, Cham, 2018. Springer International Publishing.

25 Liana Hadarean. String solving at Amazon. `https://mosca19.github.io/program/index.html`, 2019. Presented at MOSCA'19.

26 G. S. Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 32(2):147–236, 1977. (in Russian).

27 Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. Z3str4: A multi-armed string solver. In Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan, editors, *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings*, volume 13047 of *Lecture Notes in Computer Science*, pages 389–406. Springer, 2021. `doi:10.1007/978-3-030-90870-6_21`.

28 OWASP. Top 10. `https://owasp.org/www-project-top-ten/2017/`, 2017.

29 OWASP. Top 10. `https://owasp.org/Top10/`, 2021.

30 Andrew Reynolds, Andres Nötzli, Clark Barrett, and Cesare Tinelli. A decision procedure for string to code point conversion. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 218–237, Cham, 2020. Springer International Publishing.

31 Andrew Reynolds, Andres Notzlit, Clark Barrett, and Cesare Tinelli. Reductions for strings and regular expressions revisited. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 225–235, 2020. `doi:10.34727/2020/isbn.978-3-85448-042-6_30`.

32 Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.

33 Klaus U. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In Klaus U. Schulz, editor, *Word Equations and Related Topics*, pages 85–150, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

34 Z3-Noodler. Automata-based string solver, 2024. URL: `https://github.com/VeriFIT/z3-noodler`.

Antichain with SAT and Tries

Lukáš Holík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Pavol Vargovčík
Faculty of Information Technology, Brno University of Technology, Czech Republic

Abstract

We introduce a SAT-enabled version of an antichain algorithm for checking language emptiness of alternating finite automata (AFA) with complex transition relations encoded as compact logical formulae. The SAT solver is used to compute predecessors of AFA configurations, and at the same time, to evaluate the subsumption of newly found configurations in the antichain of the previously found ones. The algorithm could be naively implemented by an incremental SAT solver where the growing antichain is represented by adding new clauses. To make it efficient, we 1) force the SAT solver to prioritize largest/subsumption-strongest predecessors (so that weaker configurations are not even generated), and 2) store the antichain clauses in a special variant of a trie that allows fast subsumption testing. The experimental results suggest that the resulting emptiness checker is very efficient compared to the state of the art and that our techniques improve the performance of the SAT solver.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Regular languages

Keywords and phrases SAT, Trie, Antichain, Alternating automata, Subset query

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.15

Supplementary Material *Software (Source Code)*: `https://github.com/p4l1ly/antisat/tree/5901055e699e7b95ba3dec63d11445fcd791f5de` [50]
archived at `swh:1:rev:5901055e699e7b95ba3dec63d11445fcd791f5de`

Funding This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908 and the FIT BUT internal project FIT-S-23-8151.

1 Introduction

This paper presents a modification of a known antichain algorithm for deciding *language emptiness of alternating finite automata* (*AFA* and *AFA emptiness* for short) that utilizes SAT solving to handle complex transition relations over large structured alphabets.

AFA add conjunctive branching to nondeterminism. That is, a transition may require the rest of the word is accepted not by one successor state, but by each state in a set of successor states. This makes them exponentially more succinct than non-deterministic automata [12, 36]. The succinctness of AFA is paid for by the PSPACE-completeness of the language emptiness test. The test is based on a *de-alternation* that converts the AFA to a nondeterministic finite automaton (NFA), of at most exponential size, and checks the reachability of a final state from an initial one in the NFA. The de-alternation constructs the NFA by making explicit reachable *configurations* of the AFA and transitions between them, where a configuration is a set of all conjunctive branches possibly reached after reading some word. Efficient AFA emptiness testing algorithms mitigate the worst-case complexity by focusing on the reachability question while avoiding generating the entire NFA. They exploit the structure of the configurations to define effective state space pruning and abstraction techniques. The search is done on-the-fly, while constructing the NFA, the NFA transitions stay implicit.

AFA can be practical in applications where automata are combined with Boolean and similar operations, as demonstrated in works on string solving [1, 51], LTL model checking such as [48, 19, 26], analyzing regular expressions [15, 14], and deciding logics as WS1S [47, 25] and have a strong potential to improve efficiency in regular model checking [7, 2, 53] or deciding linear arithmetics [6, 52]. These applications indeed generate AFA with complex transition relations over large alphabets (often in the form of bit-vectors). To keep them manageable, it is necessary to work with transition relations represented symbolically.

We present a variation on the *antichain AFA emptiness test* of [22] that targets this scenario. The original antichain algorithm fights the NFA state-space explosion by pruning the reached configurations that are smaller than others (we say they are *subsumed*). This is guaranteed to preserve the answer to the reachability query. We use the variant with the backward exploration, found superior in [15, 24]. We assume that complex transition relations are represented symbolically as boolean formulae over states and symbol bits. In this case, the main bottlenecks are the computation of predecessors of already reached configurations and the pruning of predecessors that are subsumed by other already reached configurations.

Our main contribution is a way to address both these bottlenecks simultaneously and efficiently using a SAT solver. We use the SAT solver as a white box, giving us a direct access to its internal data structures and routines and to leverage their efficiency optimally. Specifically, (1) we use SAT solving with preferences [11, 21] and force the SAT solver to give priority to solutions that correspond to subsumption stronger configurations. We also encode the subsumption testing against reached configurations into the SAT query, and we boost its performance by equipping the SAT solver with a specialised data structure, based on a trie, for storing the antichain of reached configurations. We introduce novel techniques that optimize the trie. Mainly, we allow elements of every branch of the trie to be sorted in its unique way that maximizes prefix sharing (the standard trie uses a global fixed order), and we integrate into it a technique of double clause watches to facilitate fast unit propagation.

We compared an implementation of our AFA emptiness check against the best AFA emptiness checkers on a comprehensive benchmark from [24] (which compares accessible AFA checkers). The results confirm a positive impact of our new techniques and show that our AFA checker outperforms the state of the art on a significant portion of the benchmark, such as examples coming from boolean combinations of regular properties, and has a strongly orthogonal performance to the other checkers.

2 Related Work

Existing approaches to AFA emptiness and related problems [15, 34, 51, 28] adapt model-checking algorithms such as IC3/PDR [8, 33, 9] or Impact [39], or use purely automata approaches, antichain algorithms [54, 22], on which we build here, possibly enhanced with the abstraction [29] or the up-to congruence techniques [16]. Available existing implementations were recently compared in [24], showing complementary results of different tools and algorithms. We note that despite being presented as a modification of the antichain algorithm, our techniques could be applied to most of the above algorithms, and to any algorithm that generates subset-maximal predecessors/successors under a transition relation formula and prunes away those subsumed by existing configurations (also e.g. antichain algorithms for inclusion checking of NFA, tree automata, context-free grammars).

Many approaches and tools use symbolic representations of transitions, BDDs, logical formulae, or intervals of numbers [41, 35, 32, 37]. The concept of symbolic automata encapsulates a class of symbolic representations generically [17, 16]. The works [15, 34, 51] end up translating the transition relations to and-inverter graphs [40] and delegating the decision problem to a model checker. Our representation of AFA, with the transition relation described by boolean formulae over states and symbol bits, is one of the most permissive and hence compact and flexible (the other approaches require for instance that the formula is a conjunction of a part talking only about symbols and a part talking only about states, which by itself may lead to verbosity).

Among many existing SAT solvers, MiniSAT [23], on which we build, is relatively old, but still performing well on a number of real-world problems. It is a textbook example of a modern SAT solver based on the CDCL algorithm (conflict-driven clause learning) which is simple and well-documented. All modern SAT solvers are based on CDCL (except few based on deep learning [43, 3]) and share many similarities with MiniSAT, some of them [4, 44] are even founded on its source code. They differ mostly in heuristics, preprocessing, inprocessing (i.e. simplification of the problem during solving), and specialized features (like XOR support in Cryptominisat [44]). The SAT Competition 2023 was dominated by CaDiCaL and Kissat [5], which are focused on advanced heuristics and inprocessing, fine-tuned for typical SAT benchmark classes.

SATO solver [55] is the only SAT solver that used a trie-based data structure and achieved moderate speed-ups in general SAT solving. Unlike us, they did not specialise in AFA emptiness, trie in SATO were not combined with clauses and CDCL was hence harder to implement efficiently (SATO has dropped its use of tries before implementing CDCL), they also did not consider the mixed-order trie. Our implementation of watched literals also allows more utilization of the prefix sharing in the trie. The trie was also used in the CNF preprocessor of Alembic [31], to achieve reusing the preprocessor state on shared prefixes of clauses.

We use the trie primarily for the *subset query*: testing whether a given set has a superset among sets stored in a data structure. There are alternative data structures for that. We have made a preliminary comparison of the applicable data structures: LVBDDs [30], covering sharing trees [20], set tries [27], and SAT, which has indicated that the trie has the best performance and is simpler than the covering sharing trees that performed similarly.

3 Preliminaries

Boolean formulas and SAT. A *Boolean formula* ψ over a set of *variables* X is generated by the grammar $\psi ::= 0 \mid 1 \mid x \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi$ where $x \in X$. We use $\mathcal{B}(X)$ to denote the set of all Boolean formulae over X. A *positive Boolean formula* is a Boolean formula without negation and 0. The set of all positive Boolean formulae over X is denoted $\mathcal{B}^+(X)$. A *literal* is a variable or a negated variable. $m \models \varphi$ denotes that the *(boolean) assignment* $m : X \to \{0, 1\}$ *satisfies* φ (it is a *model/solution*). We will often abuse the notation and treat a boolean assignment as the set of variables assigned 1 in it (we write $x \in m$ to denote $m(x) = 1$ and relate assignments using $\subseteq$). φ is *satisfiable* iff it has a model. A formula in *CNF (conjunctive normal form)* is a conjunction of *clauses*, each being a disjunction of literals.

An instance of the *satisfiability problem (SAT)* is a CNF, and its solution is a model or the answer UNSAT if no model exists. For simplicity, we will present our algorithms in the context of the well-known Davis–Putnam–Logemann–Loveland (DPLL) method [18], even though most of the modern solvers (including ours) are based on its successor, conflict-driven clause learning (CDCL) [38, 42]. Our novel techniques are not affected by the CDCL extension.

Algorithm 1 DPLL augmented for computing preferred models [11].

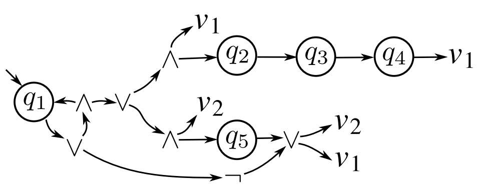
```
Input: $m$: a partial model, $P$: preferred literals, $\varphi$: immutable CNF, $\varphi_{\mathrm{mut}}$: mutable
       CNF
Output: a set $M$ of preferred models (w.r.t. $P$) of $\varphi \wedge \varphi_{\mathrm{mut}}$
1  Function SAT-PREF-REC($m$, $P$, $\varphi$, var $\varphi_{mut}$):
       // $m'$ is an updated (partial) model or CONFLICT, resp. UNSAT
2      $m' \leftarrow$ unit-propagate($m$, $\varphi \wedge \varphi_{mut}$);
3      if $m' \neq CONFLICT$ then
4          if $m'$ has an unassigned preferred literal $p \in P$ then
5              return SAT-PREF-REC($m' \cup \{p\}$, $P$, $\varphi$, var $\varphi_{mut}$) $\cup$
6                     SAT-PREF-REC($m' \cup \{\neg p\}$, $P$, $\varphi$, var $\varphi_{mut}$);
7          else
8              $m' \leftarrow$ SAT($m'$, $\varphi \wedge \varphi_{mut}$);
9              if $m' \neq UNSAT$ then
10                 $\varphi_{\mathrm{mut}} \leftarrow \varphi_{\mathrm{mut}} \wedge \bigvee_{\lambda \in P \wedge m' \models \neg\lambda} \lambda$;
11                 return $\{m'\}$;
12     return $\emptyset$;
```

SAT with preferences. We consider an extension of DPLL introduced in [11] that adds a set of *preferred literals* P as an additional input to the problem. *SAT with preferences* is a procedure `SAT-PREF(`P,φ, φ_{mut}`)` that, given a set of *preferred literals* P, the *immutable part* φ of the input formula, and the *mutable part* φ_{mut}, returns the set of *preferred models* M of the conjunction $\varphi \wedge \varphi_{\mathrm{mut}}$. The set of the preferred models consists of exactly one representative per every class of models of $\varphi \wedge \varphi_{\mathrm{mut}}$ satisfying some maximal subset of P (maximal wrt. $\subseteq$). The input formula is split into two parts because they are internally processed differently. The mutable part is iteratively updated and a specialised data structure for it is a major part of our contribution. Since it is relevant to the technicalities of our work, we recall in more detail its implementation through a recursive procedure `SAT-PREF-REC` in Algorithm 1 (it is initially called with the empty partial model m). It is a basic DPLL scheme with two differences:

1. On lines 4-6, the splitting heuristic chooses from the unassigned preferred literals first or continues with the normal SAT procedure if all preferred literals are assigned.
2. If a model is found (at line 8), it is not returned immediately. Instead, the model is added to the set of results and a clause is added (line 10) that prunes models with a subset of positive preferred literals from the further search.

Alternating automata. Let $V = \{v_1, v_2, \ldots\}$ be a set of *symbol-variables*. An assignment $a : V \rightarrow \{0, 1\}$ is a *symbol* and Σ is an *alphabet*, the set of all symbols. A *word* is a sequence of symbols and Σ^* is the set of all (finite) words.

We consider alternating automata in a form used in [49]. Symbols are encoded as bit-vectors and the transition relation for each state is given as a Boolean formula over states and symbol-bits. The solutions of the transition formula encode both the symbol as well as the set of states into which the state can transition. The formula allows to mix symbol-variables and states almost arbitrarily, the only restriction is that states do not appear under negation. This is more permissive and succinct than in other works, which typically require a separation of a formula talking about symbols from the one talking about states, and sometimes even the state formula in DNF.



$Q = \{q_1 \cdots q_5\}, \quad V = \{v_1, v_2\}, \quad \iota = q_1, \quad F = \emptyset$
$\delta(q_1) = \neg(v_1 \vee v_2) \vee (q_1 \wedge ((v_1 \wedge q_2) \vee (v_2 \wedge q_5)))$
$\delta(q_2) = q_3$
$\delta(q_3) = q_4$
$\delta(q_4) = v_1$
$\delta(q_5) = v_2 \vee v_1$

Figure 1 AFA generated from an LTL formula $((v_1 XXXv_1) \vee (v_2 X(v_2 \vee v_1)))U\neg(v_1 \vee v_2)$. Even though its set of final states F is empty, its language is not empty. A short accepting run exists, namely $\{q_1\}[v_1{=}0, v_2{=}0]\emptyset$, as the transition to the empty configuration is enabled, as the model $a \cup c_2 = \emptyset \cup \emptyset = [v_1{=}0, v_2{=}0, q_1{=}0, \cdots, q_5{=}0]$ satisfies the first operand of the top-level disjunction in $\delta(q_1)$. An example of a longer accepting run would be $\{q_1\}[v_1{=}v_2{=}1]\{q_1, q_2\}[v_1{=}v_2{=}0]\{q_3\}[v_1{=}v_2{=}0]\{q_4\}[v_1{=}v_2{=}1]\emptyset$.

Formally, an *alternating finite automaton (AFA)* is a quintuple $\mathcal{M} = (Q, V, \delta, \iota, F)$ where 1) Q is a finite set of *states*; 2) V is a finite set of *symbol-variables*; 3) $\delta : Q \to \mathcal{B}^+(Q \cup \mathcal{B}(V))$ is a *transition function*; 4) $\iota \in Q$ is the *initial state*; and 5) $F \subseteq Q$ is the set of *final states*. A *configuration* of $\mathcal{M}$ is a boolean assignment to states $c : Q \to \{0, 1\}$. The automaton has the single *initial configuration* $\{\iota\}$ and the subsets of F are *final configurations*. A configuration k *transitions* into a configuration ℓ via a symbol a, denoted as $k \xrightarrow{a} \ell$, whenever $a \cup \ell \models \bigwedge_{q \in k} \delta(q)$. We write $k \longrightarrow \ell$ if there exists a symbol a such that $k \xrightarrow{a} \ell$ and say that ℓ is a *successor* of k and that k is a *predecessor* of ℓ. We will usually use c, d for configurations in a general context and k, ℓ when speaking about a pair of a predecessor and a successor. We write $\mathsf{Pre}(a, \ell)$ and $\mathsf{Pre}(\ell)$ to denote the sets $\{k \mid k \xrightarrow{a} \ell\}$ and $\{k \mid k \longrightarrow \ell\}$ of predecessors of ℓ via the symbol a or via any symbol, respectively. A *run* of $\mathcal{M}$ over a word $a_1 \cdots a_n \in \Sigma^*$ is a sequence $c_0, a_1, c_1, a_2, \ldots, a_n, c_n$ of configurations interleaved with symbols such that for all $i : 0 \leq i < n$, $c_i \xrightarrow{a_{i+1}} c_{i+1}$. The run is *accepting* if $c_0 = \{\iota\}$ and $c_n \subseteq F$. The *language* of $\mathcal{M}$ is the set $\mathcal{L}(\mathcal{M})$ of all words with an accepting run. The *AFA emptiness problem*, that we are concerned with in this paper, is to decide whether $\mathcal{L}(M) = \emptyset$. Figure 1 has an example of an automaton with a non-empty language and its accepting run. In the rest of the paper, we will consider a fixed AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$.

Antichain algorithm for AFA emptiness. In this paragraph, we will recall the backward antichain algorithm for AFA emptiness which we optimize in this paper. AFA emptiness is well known to be a PSPACE-complete problem. It requires deciding the reachability of a final configuration from the initial one in the graph of the relation $\longrightarrow$. A *naive forward algorithm* explores the graph from the initial configuration in a search for a final one, the *naive backward algorithm* starts from the final configurations and searches for the initial configuration following $\longrightarrow$ backwards. An obvious problem is the possible exponential explosion of the search space of configurations. The antichain algorithms, first proposed in [22], introduce a search space pruning strategy based on subsumption of configurations. Antichain algorithms exist in several variants, from which we use the backward antichain. It is based on the backward naive algorithm and was found empirically superior over the forward variant in [15, 24].

Let us first outline the essence of the backward antichain informally. The *subsumption* relation used to prune the space of configuration is, in its basic variant which we use here, simply the set containement: a configuration c' *subsumes* a configuration c iff $c \subseteq c'$. The

Algorithm 2 Basic antichain algorithm.

```
   Input: AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$
   Output: Is the language of $\mathcal{M}$ empty?
 1 if $\iota \in F$ then return NOT EMPTY;
 2 $Visited \leftarrow \{F\}$;
 3 $W \leftarrow \{F\}$;
 4 while $W \neq \emptyset$ do
 5     $\ell \leftarrow \mathsf{pop}(W)$;
 6     for $a \in \Sigma$ do
 7         for $k \in \mathsf{Pre}(a, \ell)$ do
 8             if not $k \Subset Visited$ then
 9                 if $\iota \in k$ then return NOT EMPTY;
10                 $Visited \leftarrow \lceil Visited \cup \{k\} \rceil$;
11                 $\mathsf{push}(W, k)$;
12 return EMPTY
```

key property that allows to discard the configurations subsumed by other already reached configurations in antichain algorithms is that the subsumption is a simulation relation on the (reversed) relation $\longrightarrow$. The property that holds with our definition of AFA is actually even stronger: If $k \longrightarrow \ell$, $k' \subseteq k$, and $\ell \subseteq \ell'$, then $k' \longrightarrow \ell'$. This can be shown to justify the correctness of an optimized AFA emptiness test that: (1) starts the backward exploration of configuration space from only the maximal accepting configuration F, (2) keeps only the *antichain* (a set of incomparable elements) of the subsumption-maximal configurations through the search, and (3) concludes reaching the target on finding any configuration that includes ι.

Let us now recall the backward antichain algorithm for testing AFA emptiness more formally. For a set/configuration c and a set of configurations/sets A, the *subset query* of c towards A, denoted $c \Subset A$, stands for $\exists d \in A. c \subseteq d$. An *antichain* is a set of elements incomparable wrt some preorder, here a set of configurations ordered by $\subseteq$. For any set of configurations A, $\lceil A \rceil = \{c \in A \mid \neg\exists d \in A.\ c \subset d\}$ is the set of its inclusion-maximal elements.

The pseudocode of the backward antichain algorithm is in Algorithm 2. In addition to the antichain of the inclusion-maximal set of visited configurations *Visited*, it maintains a worklist W of configurations to be explored, where exploring a configuration means finding all its predecessors on lines 6-7, filtering out the subsumed ones on line 8, checking for the initialness on line 9, and updating the antichain and worklist on lines 10-11. In the paper, we focus on scenarios where the transition relation is large as a consequence of a large alphabet Σ. In such cases, Algorithm 2 has two bottlenecks.

First, the algorithm spends most of its time on line 8 with the subset query, where, for every predecessor of a visited configuration ℓ, it checks that it is not subsumed by other visited configurations stored in the antichain *Visited*. The efficiency of this subset query is obviously the primary concern since the antichain may grow exponentially large. A similar and also expensive operation is line 10, pruning the antichain from configurations subsumed by k. Line 10 is however called only a fraction of times line 8 is called (only when the test on line 8 passes), so it is much cheaper overall.

The second bottleneck is the iteration through all predecessors, $\mathsf{Pre}(\ell)$, on lines 6 and 7. We call it *predecessor iteration*. When the alphabet is large, it is often the case that $\mathsf{Pre}(a, \ell)$ and $\mathsf{Pre}(b, \ell)$ have a non-empty intersection, and the inner for-loop body is thus called many times to generate the same predecessor k.

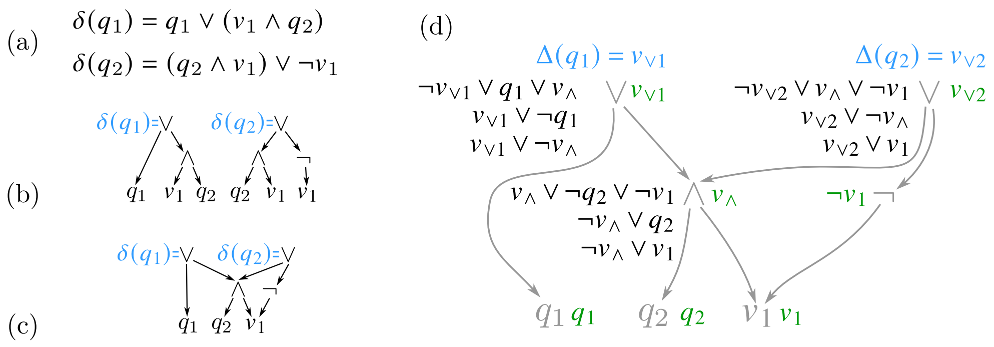


Figure 2 Representation of δ and its transformation to CNF. The transition formulae in (a) correspond to the trees in (b) where isomorphic sub-trees can be shared as in (c) and the CNF obtained by Tseyitin transformation is in (d). The links connecting states to their transition formula in (b) and (c) are shown in blue. In (d), the green literals are the names assigned by the Tseyitin transformation to the original sub-formulae (shown in grey), Δ is shown in blue, and original sub-formulae are also annotated with their equivalents produced by Tseyitin transformation. The Tseytin variables are indexed by the top-most operator of the sub-formulae they represent plus a number to distinguish between sub-formulae with the same top-most operator. Δ is depicted in blue.

This paper addresses the two bottlenecks, the subset query and the predecessor iteration, via the use of a succinct symbolic representation of the transition function and SAT solvers.

4 Symbolic Transition Function as Formula DAG

Various tools use a symbolic representation of large transition relations with large alphabets, using BDDs, intervals, formulae, effective boolean algebras, and implement the predecessor iteration efficiently in a manner that does not unfold the succinct representation. We use an alternative that is general and often very succinct. The transitions from a state are represented as a state-positive boolean formula in a compact DAG form (each single sub-formula is represented only once). The transition function δ particularly maps states to boolean formulae. The forest of abstract syntax trees of all formulae in the image of δ is a directed acyclic graph (DAG). Nodes are labelled by the function symbols $0, 1, q \in Q, v \in V, \land, \lor, \neg$ and represent subformulae, and each state q in the domain of δ is *linked* to the node representing $\delta(q)$. See Figure 2b, where links from the domain are blue. Structurally equivalent nodes are merged, see Figure 2c.

Since we are going to process the formulae in a SAT solver, we transform the forest of transition formulae into CNF using the Tseytin transformation. It is almost the standard Tseytin transformation, up to that it does not generate a Tseytin variable for the entire formula. We briefly detail it below. It generates the CNF formula $\mathsf{CNF}(\delta)$. First, the *name* of a formula φ, $n(\varphi)$, is defined as φ itself if φ is a variable, as $\neg(n(\psi))$ if φ is $\neg\psi$, and as the *Tseytin variable* v_φ if $\varphi = \psi \oplus \psi', \oplus \in \{\land, \lor\}$. In the last case, we also define a *naming formula* of φ as the CNF version of the equivalence $v_\varphi \leftrightarrow n(\psi) \oplus n(\psi')$. Finally, $\mathsf{CNF}(\delta)$ is the conjunction of all naming formulae of (con/dis)junctions that are sub-formulae in the image of δ.

The representation of δ that we will use then consists of 1) the formula $\mathsf{CNF}(\delta)$, 2) the mapping $\Delta : Q \to V_T \cup V \cup Q \cup \{\neg v \mid v \in V_T \cup V\}$ that maps each $q \in Q$ to the name of $\delta(q)$, the literal $n(\delta(q))$. An example is in Figure 2d.

5 Predecessor-subset Query with SAT Solvers

We will now show how SAT solving with preferences can be used in AFA emptiness check in a way that makes good use of the fast internal data structures of the solver, and how to further improve this by specialised data structures.

We combine the subset query and the predecessor iteration into a *predecessor-subset query*. Given a successor ℓ and an antichain of configurations *Visited*, the predecessor-subset query returns the antichain of maximal predecessors of ℓ that are not subsumed by *Visited*, that is, the set

$$\mathsf{newpre}(\ell, \mathit{Visited}) = \lceil \mathsf{Pre}(\ell) \cup \mathit{Visited} \rceil \setminus \mathit{Visited}$$

This is done by calling the procedure `SAT-PREF`$(P := \Delta(Q), \varphi, \varphi_{\mathit{Visited}})$ from Algorithm 1 on the formula that consists of the immutable part φ that represents the AFA transition relation and ℓ, and the mutable part $\varphi_{\mathit{Visited}}$ that represents the antichain *Visited*. The set of preferred literals is specified as $P := \Delta(Q)$, the variables that correspond to the predecessor states in the CNF formula $\mathsf{CNF}(\delta)$ from Section 4 representing the transition relation.

The immutable part φ is

$$\varphi := \mathsf{CNF}(\delta) \wedge \varphi_\ell \quad \text{where} \quad \varphi_\ell = \bigwedge_{q \in Q \setminus \ell} \neg q$$

After the transition relation in the first conjunct, the second conjunct φ_ℓ expresses that we want a predecessor of particularly ℓ. It asserts that after taking the transition from the predecessor, no state outside ℓ should be touched.

The mutable formula $\varphi_{\mathit{Visited}}$, used to prune the solutions subsumed by the antichain *Visited*, is

$$\varphi_{\mathit{Visited}} := \bigwedge_{c \in \mathit{Visited}} \; \bigvee_{q \in Q \setminus c} \Delta(q)$$

Here, a clause for $c \in \mathit{Visited}$ says that some state not in c must appear in a new predecessor clause, a solution to the SAT problem. The new predecessor cannot be a subset of c. The antichain *Visited* is initialised with $\mathit{Visited} := \{F\}$.

The predecessor will be read from each model m_k from the returned set of preferred models as

$$k(m_k) := \{q \in Q \mid m_k \models \Delta(q)\}$$

Note that, by the construction of the input formulae, m_k encodes precisely one transition $k \xrightarrow{a} \ell$. We extract the states of k by taking the Δ-variables, representing the pre-image, that are set to `1`. The procedure `SAT-PREF` guarantees that it will be a maximal predecessor of ℓ not subsumed by *Visited*.

The emptiness test needs to add the newly generated predecessor to the antichain. In `SAT-PREF`, this is done automatically because of the choice of $\varphi_{\mathit{Visited}}$ as the mutable part and of $\Delta(Q)$ as the preferred literals. Indeed, adding a configuration c to the antichain *Visited* represented by $\varphi_{\mathit{Visited}}$ means adding a clause with the literal $\Delta(q)$ for every q not contained in c. Let $c = k(m_k)$. Since the states q that are not contained in c are just the ones for which $\Delta(q)$ is not satisfied by m_k, adding the predecessor $k(m_k)$ to the antichain can be described by the following assignment:

$$\varphi_{\mathit{Visited}} \leftarrow \varphi_{\mathit{Visited}} \wedge \bigvee_{q \in Q \,\wedge\, m_k \models \neg\Delta(q)} \Delta(q)$$

Algorithm 3 Clausal antichain, antichain algorithm with combined subset-predecessor query.

```
   Input: AFA $\mathcal{M} = (Q, V, \delta, \iota, F)$
   Output: Is the language of $\mathcal{M}$ empty?
1  if $\iota \in F$ then return NOT EMPTY;
2  $\varphi_{Visited} \leftarrow \bigvee_{q \in Q \setminus F} \Delta(q)$;
3  $W \leftarrow \{F\}$;
4  while $W \neq \emptyset$ do
5      $\ell \leftarrow \mathsf{popMaxSize}^1(W)$ ;
6      for $m_k \in$ SAT-PREF$(\Delta(Q)$, CNF$(\delta) \wedge \varphi_\ell$, var $\varphi_{Visited})$ do
7          if $\iota \in k(m_k)$ then return NOT EMPTY;
8          push$(W, k(m_k))$;
9  return EMPTY
```

Since we call SAT-PREF with $\varphi_{\text{mut}} := \varphi_{Visited}$ and $P := \Delta(Q)$, this assignment corresponds to the update of φ_{mut} at line 10 of Algorithm 1, namely $\varphi_{\text{mut}} \leftarrow \varphi_{\text{mut}} \wedge \bigvee_{\lambda \in P \,\wedge\, m_k \models \neg\lambda} \lambda$.

The full SAT-enhanced antichain algorithm is summarized in Algorithm 3, where the predecessor-subset query covers the entire loop on line 6 of Algorithm 2, except worklist update and the initialness check. Note that SAT-PREF here indeed simultaneously finds newpre(ℓ, *Visited*), i.e., solves the subset query and the maximal predecessor iteration, and updates the antichain represented by $\varphi_{Visited}$.

We call Algorithm 3 the *clausal antichain*. A brief experiment has shown that its maximisation of predecessors via preferences is fundamental for the performance of the antichain algorithm, hence we use the feature by default in all our experiments. The lack of support for preferences in incremental SAT solvers is a major reason why one cannot just seamlessly exchange out-of-the-shelf solvers in the clausal antichain algorithm. The ease of implementing modifications such as this is one of the reasons why we are using MiniSAT.

6 Trie for Storing the Antichain

The antichain is the largest data structure that can grow exponentially in $|Q|$ and the subsumption queries over it are the most costly operations of the emptiness check. We have therefore introduced a specialised data structure based on trie for it. It replaces the internal SAT solver structures for the mutable formula, that represents the antichain. The clausal antichain equipped with a trie will be called *trie antichain*. Our trie data structure is designed to fit the following requirements:

1. a compact representation of the antichain *Visited*,
2. a fast insertion of the newly found predecessors into *Visited*, and
3. a fast subset query that integrates well with the routines and data structures of the SAT solver.

Our choice to use the trie was based on a small experiment that we did with known data structures for storing sets of sets, specialized for fast subset queries. The experiment was running unoptimized implementations of lattice-valued BDD (LVBDD) [30], set tries [27],

[1] popMaxSize is a simple heuristic, implemented previously e.g. in VATA [37], that takes a configuration with the greatest cardinality first, in a belief that its predecessors could prune a large part of the search space (our short experiment has indeed shown a performance improvement over FIFO/LIFO).

(a)

$$\begin{aligned}(v_1 \vee v_4)&\wedge\\ (v_2 \vee v_4)&\wedge\\ (v_3 \vee \neg v_5 \vee v_6)&\wedge\\ (v_3 \vee \neg v_5 \vee v_7)&\end{aligned}$$

(b)

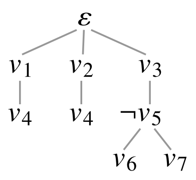


Figure 3 Clauses (a) and the corresponding trie (b) with the fixed ordering $v_1 < \cdots < v_7$.

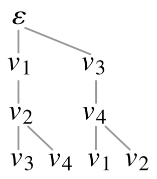


Figure 4 Trie that is smaller with a nonfixed ordering than with any fixed one.

covering sharing trees [20], and SAT unit propagation for positive boolean formulae (AND-OR graphs) with various approaches to formula simplification and sharing detection. The algorithms were solving a number of subset queries $x \subseteq Y$ with random sets. We were counting the numbers of reads and writes to the internal data structures, to get a picture of how could the optimized counterparts perform. LVBDDs solved the subset query very efficiently but their construction was unbearably prone to explosion. Set tries performed the best, closely followed by covering sharing trees (which are harder to implement). AND-OR graphs performed much worse than set tries and covering-sharing trees.

Standard trie. *Trie* is a data structure that was originally introduced in [27] for a compact representation of sets of strings. It is an ordered tree with a single root on top, denoted ε, and edges pointing downwards. The other nodes are labelled with alphabet symbols. Immediate successors of a *parent* node are its *children* and nodes without children are *leaves*. A *branch* of a trie node n is a path from n to a leaf, excluding n. A trie represents a set of strings, which are obtained by concatenating node labels of branches of ε, denoted *root branches*. Trie has been used to represent sets of sets of totally ordered elements, where a single set corresponds to a sorted string of its elements. It can thus be used to represent a CNF, viewed as a set of sets of literals. We use it to represent the antichain clauses of $\varphi_{Visited}$. An example of a trie representing the clauses from Figure 3a is shown in Figure 3b.

Mixed order trie. Sorting each branch using the same (fixed) order does not give optimal sharing of subsets as prefixes, as illustrated in Figure 4, where *no* fixed order would create as small trie as the one shown in the figure. Even if a good fixed order existed, it would be hard to guess it in advance. We therefore introduce a *mixed-order trie* where we do not fix the ordering. Allowing every branch to be sorted differently does indeed make our trie more compact. A new set x is added to the trie by finding a root branch y with a longest prefix that is a subset of x, and appending a new branch with the elements $x \setminus y$ after its prefix with the elements $x \cap y$ (with elements sorted arbitrarily). This would normally make the addition more expensive, as it would involve a search for a root branch with the largest prefix included in x. However, as we will explain in Section 6.2.2, in the context of AFA emptiness and generation of newpre(ℓ, *Visited*), the search can be completely avoided.

6.1 Clausal Watches

We adapt the well known double-watch literals scheme to our trie data structure. The scheme has been introduced in [55] and is consistently used in modern SAT solvers where it is essential for the efficiency of unit propagation and detection of conflicts. Let us recapitulate it briefly. The solver constantly watches two unassigned literals at each clause that is not yet known to be satisfied.[2] When one of the watched literals is assigned `0` and all the other unwatched literals are `0` as well, then unit propagation is triggered and assigns `1` to the other watched literal because it is the only way to satisfy the clause. When `0` gets assigned to both watched literals and all other literals in the clause are assigned `0`, then a conflict is detected. Otherwise, if other literals are unassigned or `1` on assigning `0` to a watched literal, the watch is simply moved to one of them. No action is required during backtracking where some of the watched variables may change values from `1` to `X`, keeping the only invariant that two distinct non-`0` literals are watched.

6.1.1 Double-Watch Scheme in Trie

We will detail our adaptation of the double-watch literals scheme to the clauses stored in the trie. It concerns clause prefix sharing, avoidance of repeated scanning of clauses on triggering a watch, and handling of backtracking. We will also show in Section 6.2 how our technique can work in synergy with the SAT search for preferred models, especially when it is used to generate newpre(c, *Visited*).

The double-watch technique is implemented on tries using two sets of *guards*, the *Front* and the *Rear* guards, that are essentially watches that can be assigned to nodes (and store some additional data, e.g., a pointer from *Front* to *Rear*). The SAT solver will start with one *Front* and one *Rear* at the root of the tree, and they will descend down the tree, *Rear* following *Front*. The following invariant will hold after the initialisation (described below) and after every unit propagation phase. For every root branch of the trie (clause):

1. If the root branch is not yet satisfied (does not have a `1` valued node), then it contains a single *Front* and a single *Rear*. In satisfied root branches, *Front* may be missing.
2. The *Rear* is never below the *Front*.
3. The *Rear* is at the first node valued `X` or `1` (`X` means no value assigned yet).
4. If the *Rear* is valued `X`, then the *Front* is on the second highest node that is `X` or `1`.

Initially, we have a single *Front* and a single *Rear* at the root of the trie. Note that the root is a special node without a literal. We work with it as it had the constant value `0`. The invariant (points 1-4 above) is established by *initialisation*, by calling functions `move-front-down` and then `move-rear-down` on the root. The functions implement a recursive descent of the guard watches through the tree and spreading of the guard watches to the branches, and are used to reestablish the invariant after a decision or unit propagation. The descend functions and the handlers of assigning `0` to literals are given in pseudocode in Algorithm 4.

The `move-front-down` procedure searches for a new place for *Front* in the branches of the node on which it was called. In every branch, it either

[2] The MiniSAT solver provides a watch interface where handlers can be registered to a literal. When the SAT solver valuates the watched literal with `0`, its registered handlers get triggered.

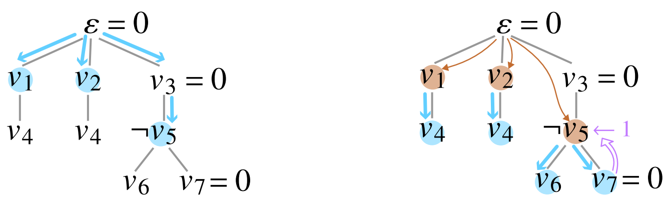


Figure 5 Initial movement of front (blue) and rear (tan) guards; unit propagation is violet.

1. places a watched copy of the *Front* at the highest node valued `X` (unassigned value) (lines 7-9), or,
2. on arrival at the leaf without seeing `X` on the way, it calls unit propagation (line 12) that assigns `1` to the literal guarded by the *Rear* above or triggers a conflict if the *Rear*'s literal is already `0`.

The `move-rear-down` procedure called on a node with a *Rear* guard works analogously, but

1. The *Front* is also sent downwards, in order to stay below the *Rear* (lines 19-22).
2. *Rear* never triggers unit propagation at leaves, as that is done when handling *Front*.

It is important to note that both `move-front-down` and `move-rear-down` are always called from a node where the respective guard is no longer present/watched. Either they are called from the watch handler, that calls line 1 or 13 and removes the watch automatically, or from line 22, which is preceded by the removal of the watch on line 21.

There are also two interesting implementation details important for efficiency. First, nodes keep a flag allowing to detect that the guard *Front* is watched on the node (on line 19), and the flag is kept up to date on lines 8 and 4. Second, a fast unit propagation on line 12 uses instant access from the *Front* to its *Rear*. The instant access is used also at line 2, to check if the root branch is not already satisfied, in which case `move-front-down` would be useless. This is facilitated by the pointer *front.rear*, which is up-to-date initially and is updated on line 20, where the *Rear* catches up with the front. Updating only at this point is indeed enough. Notice that when *Rear* moves, the connection from the *Front* guards to its copies created on line 8 will not be correctly established. It will however be reestablished in time on line 20. Indeed, the *Front* must be below the new *Rear*, hence all the literals on the way to it are valued `0`, since *Front* moves down only via `0` valued nodes. Therefore, the descend of *Rear*, which also continues uninterrupted on `0` valued nodes, will reach the *Front*, where the connection is reestablished by line 20. When *Front* is moving down, it and its copies are keeping the connection to the right *Rear* (the function `new-front` copies the pointer).

We note that in our implementation of Algorithm 4, every active *Rear* guard has a list of locations of the *Front* guards below and every *Front* guard has a pointer to its *Rear* above. This is used when *Rear* is assigned `0`, to immediately "jump" to the *Front* guards.

▶ **Example 1.** Guard movement from their initial position at ε, is shown in Figure 5. At first, *Front* guards perform a series of `move-front-down` from all nodes that have value `0` (including ε). Then, *Rear* guards are similarly transported from the root node to the nodes guarded by *Front* guards, triggering their `move-front-down` again. The node v_7, on which a front guard lands, is valuated to zero but as it is a leaf node, instead of a downward move, a unit propagation of the literal under its rear guard ($\neg v_5$) is triggered.

Algorithm 4 Implementation of Assignment Watches in a Trie.

```
Function front-watch-handler($front$):
    if value $(front.rear.literal) \neq 1$ then move-front-down($front$) ;
Function move-front-down($front$):
    $front.node.front \leftarrow NULL$;
    if $front.node$ is not a leaf then
        foreach $child$ of $front.node$ do
            if value $(child.literal) = X$ then
                $child.front \leftarrow$ new-front($front$, $child$);
                watch($\neg child.literal$, front-watch-handler, $child.front$);
            else if value $(child.literal) = 0$ then
                move-front-down(new-front($front$, $child$));
    else unit-propagate($front.rear.literal$, $front.node$);        // $front$ may cause
     conflict
Function rear-watch-handler($rear$):
    move-rear-down($rear$);
Function move-rear-down($rear$):
    foreach $child$ of $rear.node$ do
        $childRear \leftarrow$ new-rear($rear$, $child$);
        if value $(child.literal) \neq 1$ then
            if $child.front$ then
                $child.front.rear \leftarrow childRear$;
                unwatch($child.front$);
                move-front-down($child.front$);
            if value $(child.literal) = X$ then
                watch($\neg child.literal$, rear-watch-handler, $childRear$);
            else move-rear-down($childRear$) ;
```

6.1.2 Backtracking in Trie with Double Watches

In the case without the trie, there is no need for backtracking a clause state because clauses have no state. A clause is just scanned from left to right whenever its watched literal is valuated. The case with the trie is different: the watched guards must satisfy their invariant, mainly, that on root branches that are not satisfied, *Rear* and *Front* watch the highest and the second highest unassigned node and every other node above them is valued `0`. Backtracking must thus return the positions of the guards to the previous state.

This is achieved by using *guard snapshots* that are created when decisions happen. A snapshot at the decision level L consists of two vectors: 1) one capturing the state of *new guards*, those that exist after the unit propagation triggered by the decision terminates but did not exist before the decision, 2) and one capturing the state of *old guards*, those that existed before but not after L. When the decision L is undone by backtracking, the new guards get removed and the old guards get reestablished.

As depicted in Figure 6, both lists of guards are empty when a new decision level is enetered and get updated when guards move downwards within the decision level. Namely, the functions `move-front-down` and `move-rear-down` would additionally call a procedure that updates the snapshot. The first move of a guard within the decision level adds its position to the list of old guards, and every further move replaces the guard in the list of new guards by its descendants received by the children of the current node.

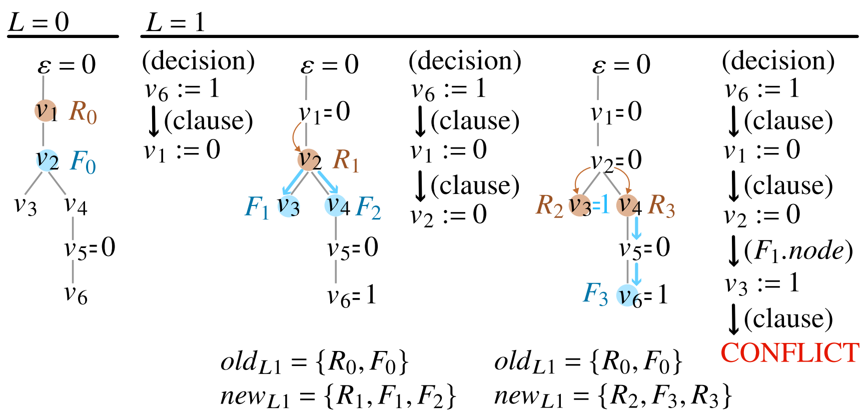


Figure 6 Backtracking. On the left, the position of *Rear* R_0 and *Front* F_0 at the end of level 0 ($L0$) is shown. Then, at the start of L1, the assignment $v_6 := 1$ is decided, from which an external clause unit propagates $v_1 := 0$. This triggers the movement of R_0, which further triggers the movement of F_0. As R_0 and F_0 were not created in the current level L1, they are added to the list of old guards. The list of new guards is updated with R_1, F_1, F_2. The unit propagation continues and another external clause unit propagates $v_2 := 0$, which triggers further guard movement. The moved guards have been created in L1, therefore, old_{L1} is untouched. The moved guards are replaced in new_{L1} with their descendants. As F_1 was moved down from the leaf node, it caused a unit propagation $v_3 := 1$ at its rear R_2. The consequent unit propagation raises a conflict. If we were in the context of CDCL, a conflict analysis follows, during which the node of the former F_1 is queried for the reason of the propagation. The reason contains negations of 0-valued literals of the root branch that ends in the queried leaf node, namely $\neg v_1 \wedge \neg v_2$. Backtracking from L1 recreates the guards from old_{L1} and deletes the guards listed in new_{L1}, which reverts the trie to the state at the end of L0.

This mechanism can be implemented fairly efficiently by equipping every guard with a pointer to its place in the snapshot. The pointer allows a constant-time quick removal/replacement of the guard from the snapshot and an addition of a guard to the snapshot is constant time.

6.2 Constructing a Trie

A SAT solver can use the trie in two ways. A generic mode that can be used in general SAT solving can be optimized and fine-tuned to fit the generation of newpre(c, *Visited*) in the AFA emptiness check. We will first comment on the generic mode and then contrast the AFA specialised mode against it.

6.2.1 Constructing a Trie For General SAT Solving

The set of clauses in the input of a SAT solver is added to the trie in a standard way. For the input clauses to be added to the trie, we first need to pick an ordering of literals and sort the clauses (a good option is for instance ordering by the frequency of literals in the input clauses).

A clause is then added by traversing the trie from the root (ε) down, simultaneously with the added clause. The child into which the traversal descends is chosen as the one with the ith literal of the clause. If the child with the ith literal is not present, the descend ends and the remaining literals of the clause are added as a new branch of the current node.

▶ **Example 2.** Let us add the clause $v_1 \vee v_3 \vee \neg v_5$ into the trie from Figure 3 b). The first literal v_1 of the clause is found among the children of the root. The second one is not found among the children of the node v_1, so the new branch $v_3, \neg v_5$ is added under the node v_1.

In case that all literals of the added clause are exhausted (i is larger than the length of the clause), the descend also ends and the entire subtree beyond the $(i-1)$th visited node is removed (rather all the sub-trees rooted by the children of the $(i-1)$th node). Indeed, the removed root branches represented clauses strictly larger than the added clause, hence they were redundant (by boolean absorption).

▶ **Example 3.** Adding the clause v_3 into the trie from Figure 3b would remove the nodes $\neg v_5, v_6, v_7$. Note that not all boolean absorption is detected this way – it is highly dependent on the order of clauses. E.g., the clause v_4 would be created as a new branch of ε and the absorbed branches v_1, v_4 and v_2, v_4 would remain in the trie.

6.2.2 Constructing a Trie for AFA

In the AFA emptiness check, we optimize handling of the trie for the particular case of generating newpre(c, *Visited*). We use the mixed-order trie, with root branches representing clauses ordered arbitrarily. The flexible ordering of root branches allows to maximize sharing of prefixes and hence to reduce the size of the trie. Normally, the cost of allowing the mixed order would be heavy. Whenever adding a clause, we would have to perform an expensive depth-first search for a branch with a longest prefix included in the clause. However, as we will explain below, due to the particular way we are using the trie in generating newpre(c, *Visited*) and the particular way of how guards are implemented (Section 6.1.1), an addition of a clause is cheap and requires no search at all.

Let us now describe the mechanism in detail. Recall that root branches are clauses of the formula $\varphi_{Visited}$. The search for a new predecessor starts in a state in which the previous SAT call finished with a model m_k that encodes the newly found predecessor k such that for each $q \in Q$, $(m_k \models \Delta(q)) \leftrightarrow q \in k$.

Adding a predecessor to the antichain. To find the next predecessor, we need to add k to *Visited* and restart the SAT solver, preferably in a way that reuses as much of the previous computation as possible. Recall that adding k to *Visited* means adding to the trie the clause

$$\psi_k = \bigvee_{q \in Q \setminus k} \Delta(q) = \bigvee_{q \in Q \,\wedge\, m_k \models \neg\Delta(q)} \Delta(q)$$

As discussed in Section 6.1.1, the SAT solver finished in a state where on every branch π, the *Rear* guard is on the first non-0-valued literal, and everything in the prefix $\bar{\pi}$ above it is valued 0.

▶ **Lemma 4.** *On every branch π, the prefix $\bar{\pi}$ ending at the parent of the rear guard is the largest prefix valued only by zeroes.*

Hence, on every branch π, the path $\bar{\pi}$ from the root (excluded) to the parent of the rear guard is the longest prefix included in ψ_k We can therefore add ψ_k efficiently to the trie as follows. We chose a root branch π with a deepest rear guard. Thus, π is a root branch with a largest prefix, $\bar{\pi}$, included in ψ_k. We append elements of $\psi_k \setminus \bar{\pi}$ as a new branch after $\bar{\pi}$. We call it *newly added branch.*
(We abuse the notation and implicitly convert between sequences of nodes and sets of literals. For example, $\psi_k \setminus \bar{\pi}$ above stands for the branch that arises from the branch ψ_k by removing the nodes with literals of the path $\bar{\pi}$.)

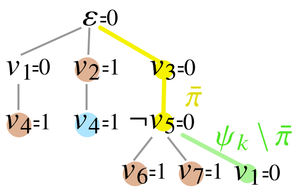


Figure 7 Adding a new clause $\psi_k = v_1 \vee v_3 \vee \neg v_5$ to a trie. *Rear*/*Front* guards are tan/blue, the largest prefix included in ψ_k is yellow, the added branch is green.

▶ **Example 5.** In Figure 7, we are adding a clause $\psi_k = v_1 \vee v_3 \vee \neg v_5$ to the trie. The deepest rear guards are v_6, v_7, we select one of them arbitrarily, v_6. The path from ε to the parent of v_6 is $\bar{\pi} = v_3, \neg v_5$. It is a longest prefix of the root branches that is included in the set ψ_k. The branch $\psi_k \setminus \bar{\pi} = v_1$ is added to the end node of $\bar{\pi}$. Note that 1) the new root branch $v_3, \neg v_5, v_1$ is fully valuated with 0, hence conflicting; and 2) if we used the fixed order $v_1 < \cdots < v_7$ instead, we would add a longer branch $v_3, \neg v_5$ under the node v_1.

Restarting the SAT solver. With a clause k added to the antichain, the next call of the SAT solver that finds the next maximal predecessor can reuse much of the previous state. First, note that after finding m_k, addition of the branch ψ_k will cause a conflict exactly in ψ_k (since all its literals are 0 in m_k). We may reuse much of the solver's state by just starting backtracking from the conflict in ψ_k. We need to ensure that after the backtracking, the guard invariant from Section 6.1.1 will be satisfied. These invariants would hold in common backtracking during SAT solving, but as we have added a new branch $\psi_k \setminus \bar{\pi}$, we have to enrich the snapshots of the guard history with guards in the newly added branch, as if it always existed. The snapshots of the guards are reconstructed using a simulation of the decisions that led to constructing m_k.

The simulation uses decision levels of literals in the new root branch to determine positions of *Rear* and *Front* in each decision level. It is based on the property that whenever a guard movement is triggered in a level L, the guard moves down through all nodes valuated with a lesser or equal level to L, because they already have the value 0, and it stops at the first node with a higher level than L, which is still unassigned, or raises a conflict if such node does not exist. The guard may also temporarily stop at nodes with the same decision level L, as they might not be valuated yet, but it will continue from them later in L (we are interested only in the position of the guard at the end of the decision level L).

For efficient backtracking in the newly added branch, we sort it by the decision levels of the literals. This way, during the backtracking, the guards in the branch will simply climb up.

▶ **Example 6.** Before describing the general algorithm, this example shows an intuitive reconstruction of the guard history in a newly added branch. Figure 8 shows a new root branch ψ_k, with the newly added branch $\psi_k \setminus \bar{\pi}$ starting at its fifth node v_5. Literals at nodes are not used in the reconstruction (we only know that they all have values 0, and, consequently, the deduction always ends with a conflict). The reconstruction is based solely on decision levels when literals were valuated in the previous SAT solving. The decision levels are shown on the left side of the nodes. At the right side, we can see the results of the reconstruction – deduced positions of *Rear* and *Front* guards at the end of levels when they moved.

We start the example at the end of level 1, when *Rear* is at v_1 and *Front* is at v_3 (we skip explanation of why the guards are at those positions, anyway, before level 3, the guard positions are irrelevant for enriching snapshots, as no guards are present in the newly added branch).

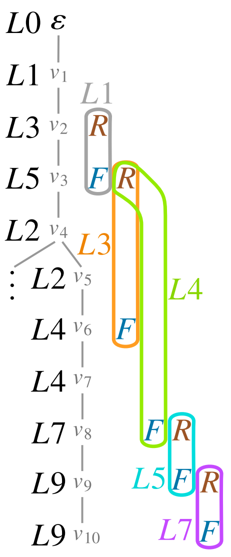


Figure 8 Reconstructing the history of guards in a new branch from decision levels of variable assignments.

During level 3, node v_2 is assigned `0` and the *Rear* triggers the downward move of the *Front*, which stops as far as at node v_6, as it is the first node that is assigned after level 3. The *Front* is then moved two steps down during level 4, because of `0`-assignment to the literals of the two nodes. In level 5, `0` is assigned to the literal under *Rear*, and *Rear* therefore moves down to *Front*, triggering the move of *Front* to the next node. In level 7, another `0` assignment moves *Rear* one step down, which again moves *Front* one step down. In level 9, both guards move from their positions but raise a conflict.

Let us generalize the deduction of guard positions from the previous example in a *deduction algorithm.*
The input to the deduction is the new root branch ψ_k, partitioned to the shared prefix $\bar{\pi}$ and the newly added branch $\psi_k \setminus \bar{\pi}$. The root branch ψ_k is ordered as it is present in the trie – the new suffix $\psi_k \setminus \bar{\pi}$ is ordered by the decision levels of its literals, but it may not be the case for the shared prefix $\bar{\pi}$.
The output of the deduction are two *movement sequences*, one for *Rear* the other for *Front*. A movement sequence is either an empty sequence (if the guard does not visit the newly added branch before the conflict level) or it is an alternation $L_1, N_1, L_2, N_2, \ldots, N_{n-1}, L_n$ of levels L_i and such nodes N_i, onto which the guard moved in L_i. The level L_1 is when the guard enters the newly added branch and L_n is the level of the conflict.

1. First, we find the level L_{R0}, when the *Rear* enters the newly added branch. It is the greatest level of the shared prefix $\bar{\pi}$, or 0 if $\bar{\pi}$ is empty.
2. Then, we find the node N_{R0}, where *Rear* resides at the time when *Front* enters the newly added branch (*Rear* stays at that node as long as until L_{R0}). It is the topmost node with L_{R0}, possibly ε.
3. Next, we find the level L_{F0} when *Front* enters the newly added branch. It is the highest level of $\bar{\pi} \setminus \{N_{R0}\}$, or 0 if $|\bar{\pi}| \leq 1$.
   This can be explained as follows. *Rear* visits N_{R0} in the highest level above N_{R0} (or in level 0 if N_{R0} is the first in π), let us call that level L_{push}. It immediately triggers the

downward move of *Front*. The *Front* then continues its movement in the suffix of $\bar{\pi}$ below N_{R0} without being pushed down by *Rear* and exits $\bar{\pi}$ in the highest level of the suffix (or immediately if the suffix is empty or L_{push} is higher).

4. Next, we record *Front* movements in the levels L such that $L_{F0} \leq L < L_{R0}$ using the following loop. The loop starts with $L \leftarrow L_{F0}$ and ends when $L_{R0} \leq L$ (which may hold initially). The following is the loop body. To the movement sequence of *Front*, we append a movement L, N where N is the topmost node of $\psi_k \setminus \bar{\pi}$ that has a greater level than L. If there is no such node, conflict is reached and the whole deduction is finished. Otherwise, we set L to the level of that node and repeat.
5. In the levels L such that $L_{R0} \leq L$, both *Rear* and *Front* move in the newly added branch. *Rear* moves in the same way as *Front* moved in the previous point. *Front* is always just one node below *Rear*, which follows from the ascending order of decision levels in the newly added branch. If *Rear* gets to the leaf, *Front* cannot be lower, conflict is reached and the deduction is done.

After the movements of the two guards in the relevant levels are deduced, snapshots are enriched in an obvious way. Namely, for all levels where a guard moves down (or causes a conflict), 1) the guard is added to the list of old guards if the move started in the newly added branch, 2) the moved copy of the guard is added to the list of new guards if it lands in the newly added branch.

▶ **Example 7.** In Figure 8, the *Front* movement sequence is $L3, v_6, L4, v_8, L5, v_9, L7, v_{10}, L9$ and the *Rear* movement sequence is $L5, v_8, L7, v_9, L9$. In level 3, *Front* has moved into the newly added branch from outside and landed at node v_6. Therefore, only the list of new guards of L3 is enriched with *Front* at that node. In level 4, the old (v_6) and new (v_8) position of *Front* is added to the old and new list respectively. In level 5, *Rear* is coming from outside, so its position at v_8 is added only to the new guards, while both old (v_8) and new (v_9) positions of *Front* are added to the respective lists. In level 7, both lists are enriched with old/new positions of both guards. In level 9, both *Rear* at v_9 and *Front* at v_{10} are added only to the list of old guards, as they started to move but a conflict was raised.

Initialization of the trie. The trie is initialized with the starting configuration $Q \setminus F$ of the backward search represented as the clause $\bigvee_{q \in Q \setminus F} \Delta(q)$ with an arbitrary ordering of the literals, and both guards are placed at the root.

7 Experiments

Implementation. We have implemented the clausal and the trie antichain in the tool AntiSAT, in C++, as a modification of MiniSAT [46], which we have chosen as a competent and modern solver, yet still simple and well documented enough to be modified with a reasonable effort. To implement the trie, we needed the version 1.12b, which is the last version that supports other constraints than clauses. We have however ported back major enhancements of the newer versions (except improvements of preprocessing), namely `StrengthenCC` [45]. We have implemented the trie data structure with memory locality and cost of allocations in mind: siblings are stored in a contiguous vector; added branch is allocated in a single array; guard data are present just next to the data of the guarded node. For SAT problems, the trie does not change during the solving, so we allocate a single memory region where the whole trie is moved after the construction.

Evaluation of AFA emptiness checking. The main focus of our experiment is to evaluate our AFA emptiness checker against other checkers, and evaluate the impact of using trie on the performance. We do not present a detailed evaluation of predecessor maximisation. Without it, our solver was not competitive at all, hence we include it to the baseline.

We have used the benchmark set from [24] with the following modifications. We have regenerated AFA from LTL because there was an unnecessary removal of final states applied previously. Additionally, to mitigate the effect of obvious redundancies in automata structure, we have applied simple preprocessing consisting of basic structural subformula sharing detection, merging of states with the same subformula and the same finalness, removing structurally unreachable states and their transitions, removing variables that occur only positively or only negatively, and applying few boolean laws: double negation, idempotence, annulment, absorption and complement law.[3]

We have compared the two variants of AntiSAT (with and without a trie) with the two most successful tools from [24]: ABC [10] (using IC3/PDR) and Mata [13] (Mata does not check AFA but boolean combinations of NFA, so it is not applicable to all benchmarks). All solvers and preprocessing tools are run with a time limit of 60 seconds and a memory limit of 10GiB. The 60s seem sufficient and extending the time limit does not seem to influence the overall comparison of tools significantly. Memory errors (which are very rare) are displayed together with timeouts. The experiments were run on a machine with 12th Gen Intel(R) Core(TM) i7-1260P CPU, running Linux. The results can be seen in Table 1 and Figure 9. The following can be concluded:

1. The trie in AntiSAT improves solving times (by a factor of 2 to 11 on average), except on "automata inclusion", where it is slightly slower (but Mata is here much faster anyway).
2. ABC and Mata are both complementary to AntiSAT. AntiSAT shines in "bool comb" benchmarks, Mata is very good at solving "automata inclusion", while ABC wins at "stranger afa", "ltl afa", and "noodler" benchmarks. There are several exceptions though.
3. The "email filter" benchmarks are easy for all solvers and most of the "noodler" benchmarks too.

The main positive takeout is that AntiSAT solves the entire "bool comb" and many examples from "ltl afa" and "noodler" much faster than the others, and it is strongly orthogonal to the other tools.

Regarding the impact of the trie data structure, Table 2 compares numbers of literals in the clausal and the trie representation of *Visited* (i.e. the number of trie nodes for AntiSAT[trie], or the total number of literals in all clauses of φ_{mut}) at the end of solving an AFA instance. In fact, for the "bool comb" benchmark, where the trie had the greatest effect in terms of solving time, less prefix sharing is shown than for other benchmarks (but still saving 500 thousand of literals on average).

Figure 10 shows the solving of three chosen nontrivial benchmark instances: the time that was spent on each predecessor query, the overall number of predecessor queries, and the results of the queries. Trie and clausal AntiSAT are compared. In the "bool comb" benchmark, clausal SAT spends a lot of time on finding predecessors (the SAT queries that find a model), while final proofs that there are no more predecessors (when the query results in UNSAT) are very fast. Trie is fast in both cases. The other two benchmarks are different: SAT queries with the UNSAT result are usually slower than queries that found predecessors. In the last benchmark, from "automata inclusion", the trie is slower than clausal AntiSAT. The performance of the trie does not seem to correlate with the number of states or symbol-variables.

[3] The instances are available at `https://github.com/p4l1ly/antisat-afa-benchmarks` at tag `sat2024`.

Table 1 Statistics from the solver run-times. $|B|$ is the number of instances in the benchmark, S_1 is the number of benchmarks solved only by the first solver of the two in the column, S_2 only by the second one, S_{12} by both. T_i is the average time the first/second solver has taken to solve interesting instances. An instance is *interesting* if both solvers solved it within 60 seconds and at least one solver took more than 1 second (to have a comparison of run-times not cluttered by trivial instances).

		AntiSAT[clause] x AntiSAT[trie]					ABC x AntiSAT[trie]					Mata x AntiSAT[trie]				
benchmark	$\|B\|$	S_1	S_2	S_{12}	T_1	T_2	S_1	S_2	S_{12}	T_1	T_2	S_1	S_2	S_{12}	T_1	T_2
automata inclusion	136	0	0	123	6.3	9.0	4	9	114	10.3	7.4	12	0	123	1.0	8.7
bool comb	653	0	6	540	11.6	1.0	0	78	468	14.3	1.6	6	97	449	4.1	2.2
email filter	500	0	0	500	NaN	NaN	0	0	500	7.3	0.2	0	0	500	1.2	0.1
ltl afa	7087	0	19	5724	14.9	7.9	1161	58	5685	0.3	14.2	Mata not applicable				
noodler	13840	0	6	13790	9.0	5.6	42	14	13782	0.4	9.9	Mata not applicable				
stranger afa	4058	0	291	3131	31.9	18.1	597	0	3422	0.7	34.0	Mata not applicable				

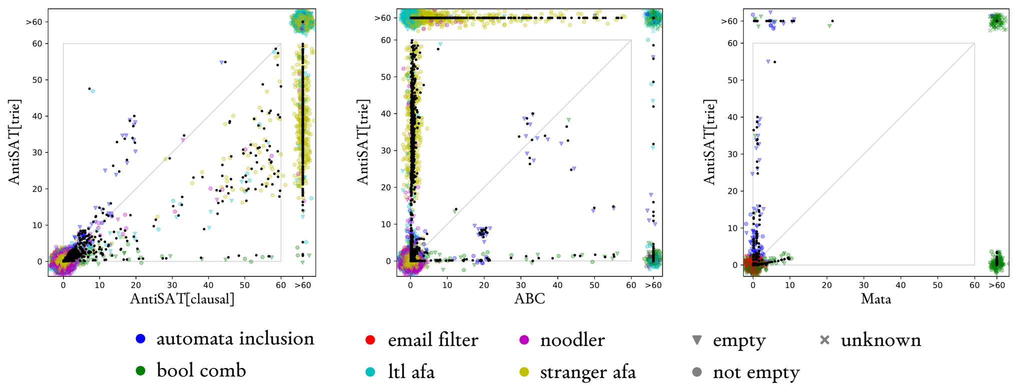


Figure 9 Detailed comparison of pairs of AFA emptiness checkers. Exact times are marked with black dots but to give an idea about the number of overlapping points, there are semitransparent associated points with small random shifts. The colour of the associated points indicates the benchmark set, while the shape indicates the answer (empty or not).

Table 2 Statistics of sizes of the clausal and trie representation of *Visited* taken from the interesting instances of the comparison AntiSAT[clause] x AntiSAT[trie]. $|B_I|$ is the number of the interesting instances in the benchmark. The abbreviations SzC/SzT mean size (i.e., the total number of literals) of clausal/trie representation of *Visited*.

		SzC / 1000			SzT / 1000		
benchmark	$\|B_I\|$	min	max	avg	min	max	avg
automata inclusion	86	108	5792	2509	51	1019	456
bool comb	60	182	19360	1510	65	18727	1065
ltl afa	62	244	60569	4145	8	15910	540
noodler	62	88	4605	1169	19	1470	245
stranger afa	87	270	32892	10110	5	622	312
total	357	88	60569	4245	5	18727	501

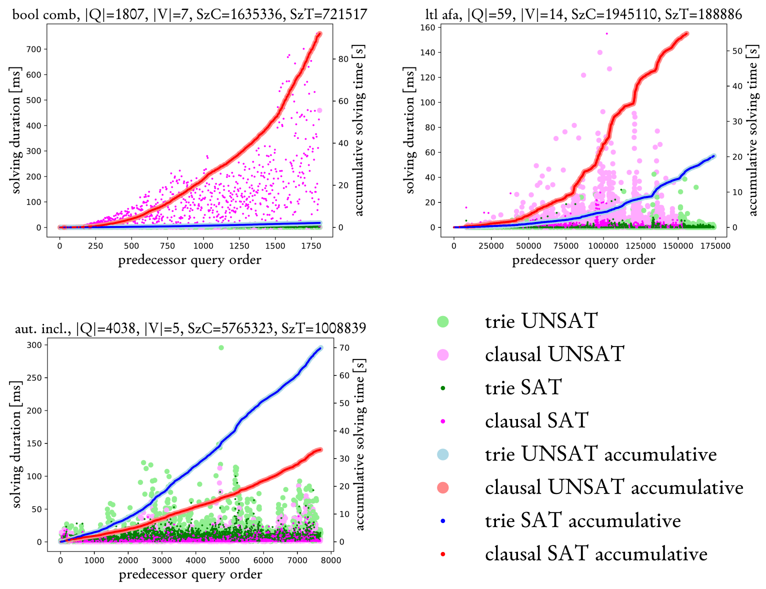


Figure 10 A detailed insight into solving three emptiness tests. Each graph shows SAT solving times throughout the solving of AntiSAT[clausal] and AntiSAT[trie] – the individual solving times and their accumulation.

8 Conclusion and Future Directions

We have proposed an AFA emptiness check that uses a SAT solver to generate a maximal predecessor not subsumed by the set of visited configurations. Our techniques performed significantly better than the state-of-the-art on a large portion of benchmarks. The same techniques can most probably be used also in the IC3-based AFA emptiness check (which can be seen as an augmentation of the backward antichain with a form of an abstraction refinement). Application of our techniques in IC3 may potentially be even more interesting than the presented adaptation to the backward antichain. A question is whether tries could be implemented in some top-performing SAT solver as CaDiCaL and have an impact in general SAT solving or whether a faster base solver would improve the performance of the AFA emptiness check.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Bui Phi Diep, Julian Dolby, Petr Janků, Hsin-Hung Lin, Lukáš Holík, and Wei-Cheng Wu. Efficient handling of string-number conversion. In *Proc. of PLDI'20*, pages 943–957. ACM, 2020.

2 Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*, pages 116–130. Springer, 2002.

3 Saeed Amizadeh, Sergiy Matyunin, Helmut Weimer, and Marco Maratea. Learning combinatorial optimization algorithms over graphs. In *Proc. of NeurIPS'18*, volume 31, 2018.

4 Gilles Audemard and Laurent Simon. Glucose: A solver that predicts learnt clauses quality. In *SAT Competition*, 2009.

5 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In *SAT Competition*, 2020.

6 Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: an overview. In *Proc. of ICLP'02*, volume 2401 of *LNCS*, pages 1–19. Springer, 2002.

7 Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *Proc. of CAV'20*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

8 Aaron R Bradley. SAT-based model checking without unrolling. In *Proc. of VMCAI'11*, pages 70–87. Springer, 2011.

9 Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Proc. of FMCAD'07*, pages 173–180. IEEE Computer Society, 2007.

10 Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Proc. of CAV'10*, pages 24–40. Springer, 2010.

11 Thierry Castell, Claudette Cayrol, Michel Cayrol, and Daniel Le Berre. Using the Davis and Putnam procedure for an efficient computation of preferred models. In *Proc. of ECAI'96*, volume 96, pages 350–354. Citeseer, 1996.

12 Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

13 David Chocholatý, Tomáš Fiedor, Vojtěch Havlena, Lukáš Holík, Martin Hruška, Ondřej Lengál, and Juraj Síč. Mata: A fast and simple finite automata library. In *Proc. of TACAS'24*, pages 130–151. Springer, 2024.

14 Arlen Cox. Model Checking Regular Expressions. URL: `https://mosca19.github.io/slides/cox.pdf`, 2019. Presented at MOSCA'19.

15 Arlen Cox and Jason Leasure. Model checking regular language constraints. *CoRR*, abs/1708.09073, 2017. `arXiv:1708.09073`.

16 Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *Electronic Notes in Theoretical Computer Science*, 336:79–99, 2018.

17 Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. *ACM SIGPLAN Notices – POPL'14*, 49(1):541–553, 2014.

18 Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

19 M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *Proc. of TACAS'08*, pages 63–77. Springer, 2008.

20 Giorgio Delzanno and Jean-François Raskin. Symbolic representation of upward-closed sets. In *Proc. of TACAS'00*, pages 426–441. Springer, 2000.

21 Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. Solving satisfiability problems with preferences. *Constraints*, 15:485–515, 2010.

22 Laurent Doyen and Jean-François Raskin. Antichain algorithms for finite automata. In *Proc. of TACAS'10*, LNCS. Springer, 2010. `doi:10.1007/978-3-642-12002-2_2`.

23 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. of SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

24 Tomáš Fiedor, Lukáš Holík, Martin Hruška, Adam Rogalewicz, Juraj Síč, and Pavol Vargovčík. Reasoning about regular properties: a comparative study. In *Proc. of CADE'23*, pages 286–306. Springer, 2023.

25 Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. Lazy automata techniques for WS1S. In *Proc. of TACAS'17*, volume 10205 of *LNCS*, pages 407–425. Springer, 2017.

26 Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

27 Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

28 Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In *Proc. of TACAS'13*, LNCS. Springer, 2013. `doi:10.1007/978-3-642-36742-7_20`.

29 Pierre Ganty, Nicolas Maquet, and Jean-François Raskin. Fixed point guided abstraction refinement for alternating automata. *Theory of Computer Science*, 411(38-39):3444–3459, 2010.

30 Gilles Geeraerts, Gabriel Kalyon, Tristan Le Gall, Nicolas Maquet, and Jean-Francois Raskin. Lattice-valued binary decision diagrams. In *Proc. of ATVA'10*, pages 158–172. Springer, 2010.

31 Hyojung Han and Fabio Somenzi. Alembic: an efficient algorithm for CNF preprocessing. In *Proc. of DAC'07*, DAC '07, pages 582–587. ACM, 2007.

32 Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proc. of TACAS '95*, volume 1019 of *LNCS*. Springer, 1995.

33 Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proc. of SAT'12*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

34 Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL), 2018.

35 Peter Kelb, Tiziana Margaria, Michael Mendler, and Claudia Gsottberger. MOSEL: A sound and efficient tool for m2l(str). In *Proc. of CAV '97*, volume 1254 of *LNCS*, pages 448–451. Springer, 1997. `doi:10.1007/3-540-63166-6_45`.

36 Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not that weak. *ACM Trans. Comput. Logic*, 2(3):408–429, 2001.

37 Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *Proc. of TACAS'12*, volume 7214 of *LNCS*. Springer, 2012. `doi:10.1007/978-3-642-28756-5_7`.

38 João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, May 1999.

39 Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

40 Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *Proc. of DAC '06*, pages 532–535. ACM, 2006.

41 Anders Møller et al. Brics automata library. URL: `https://www.brics.dk/automaton/`.

42 M. W. Moskewicz, L. Zhang, C. F. Madigan, S. Malik, and Y. Zhao. Chaff: Engineering an efficient sat solver. In *Design Automation Conference*, pages 530–535. IEEE Computer Society, 2001.

43 Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *Proc. of ICLR'19*, 2019.

44 Mate Soos. Enhancing CDCL SAT solvers with extended resolution. In *Proc. of Pragmatics of SAT*. EasyChair, 2010.

45 Niklas Sörensson and Niklas Eén. Minisat – A SAT solver with conflict-clause minimization. *Proc. of SAT'05*, pages 1–2, 2005.

46 Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *Proc. of SAT'09*, page 31, 2009.

47 Dmitriy Traytel. A coalgebraic decision procedure for WS1S. In *Proc. of CSL'15*, volume 41 of *LIPIcs*, pages 487–503. Schloss Dagstuhl, 2015.

48 Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. of LICS'96*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

49 Pavol Vargovčík and Lukáš Holík. Simplifying alternating automata for emptiness testing. In *Proc. of APLAS'21*, pages 243–264. Springer, 2021.

50 Pavol Vargovčík. AntiSAT. Software, version 0.1.0., Czech Ministry of Education, Youth and Sports ERC.CZ LL1908 FIT BUT FIT-S-23-8151, swhId: `swh:1:rev:5901055e699e7b95ba3dec63d11445fcd791f5de` (visited on 2024-07-31). URL: `https://github.com/p4l1ly/antisat/tree/5901055e699e7b95ba3dec63d11445fcd791f5de`.

51 Hung-En Wang, Tzung-Lin Tsai, Chun-Han Lin, Fang Yu, and Jie-Hong R. Jiang. String analysis via automata manipulation with logic circuit representation. In *Proc. of CAV'16*, volume 9779 of *LNCS*. Springer, 2016.

52 Pierre Wolper. On the use of automata for deciding linear arithmetic. In *Proc. of TABLEAUX'09*, volume 5607 of *LNCS*, page 16. Springer, 2009.

53 Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, pages 88–97. Springer, 1998.

54 Martin De Wulf, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.

55 Hantao Zhang and Mark Stickel. Implementing the Davis–Putnam method. *Journal of Automated Reasoning*, 24(1):277–296, 2000.

Towards Universally Accessible SAT Technology

Alexey Ignatiev
Monash University, Melbourne, Australia

Zi Li Tan
Monash University, Melbourne, Australia

Christos Karamanos
Monash University, Melbourne, Australia

Abstract

Boolean satisfiability (SAT) solvers are a family of highly efficient reasoning engines, which are frequently used for solving a large and diverse variety of practical challenges. This applies to multidisciplinary problems belonging to the class NP but also those arising at higher levels of the polynomial hierarchy. Unfortunately, encoding a problem of user's interest to a (series of) propositional formula(s) in conjunctive normal form (CNF), let alone dealing with a SAT solver, is rarely a simple task even for an experienced SAT practitioner. This situation gets aggravated further when the user has little to no knowledge on the operation of the modern SAT solving technology. In 2018, the PySAT framework was proposed to address the issue of fast and "painless" prototyping with SAT solvers in Python allowing researchers to get SAT-based solutions to their problems without investing substantial time in the development process and yet sacrificing only a little in terms of performance. Since then, PySAT has proved a useful instrument for solving a wide range of practical problems and is now a critical package for the PyPI infrastructure. In the meantime, there have been advances in SAT solving and enhancements to PySAT functionality to extend its modelling and solving capabilities in order to make modern SAT technology accessible and deployable on a massive scale. This paper provides a high-level overview of the current architecture of PySAT and some of its capabilities including arbitrary Boolean formula manipulation, CNF preprocessing, and support for external user-defined propagators.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories; Hardware → Theorem proving and SAT solving; Theory of computation → Constraint and logic programming

Keywords and phrases PySAT, Python, Prototyping, Practical Applicability

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.16

Supplementary Material *Software (PySAT Homepage)*: `https://pysathq.github.io/`

1 Introduction

Conflict-driven clause learning (CDCL) SAT solving serves as an illustrious example of a success story in Computer Science [32, 35, 36, 39, 58, 18, 19, 17, 42, 5, 4, 28, 9], providing a family of highly efficient decision oracles usable for solving myriads of practical problems. On the other hand, implementing solutions to practical problems based on the state-of-the-art SAT technology often requires one to be a SAT expert, which hampers the widespread use of SAT and its generalizations. Addressing the above issue was one of the motivations behind the proposal of the PySAT framework [24] designed specifically to ease incremental SAT-based prototyping. Since its inception, PySAT has become a valuable everyday instrument widely used in practice for tackling various AI problems. In 2021, based on the daily downloads statistics, PySAT has been included in the list of top-1% PyPI (Python Package Index) packages [48, 49] being named one of the critical projects for PyPI infrastructure.

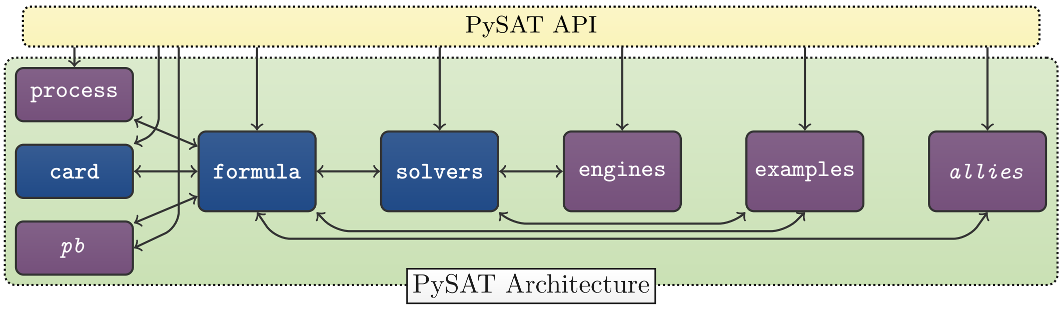


Figure 1 PySAT framework. Modules `solvers`, `formula`, and `card` shipped with the original version [24] appear in blue. Modules providing access to third-party tools appear in italics.

Since the release of PySAT, there have been numerous enhancements made both in the framework and in the state of the art of SAT solving in general. Hence, the original paper [24] no longer serves as an up-to-date account of PySAT's capabilities. To fill in this gap, this paper reviews the current functionality of PySAT, focusing on its architecture and a few novel components, namely, arbitrary Boolean formula manipulation, CNF formula (pre-)processing and the capacity to accommodate external reasoning engines following the recent IPASIR-UP interface [20]. The paper also experiments with the latter demonstrating that it may be feasible and favorable to implement external propagators in Python as it appears to be a good trade-off between the development challenges and the overall solver performance.

2 Framework Architecture

Originally, PySAT comprised three core modules: `solvers`, `formula`, and `card` – providing access to state-of-the-art CDCL SAT solvers [34], CNF formula manipulation, and cardinality constraint encodings [50], respectively. Since then the list of modules expanded and now additionally includes modules `process` for formula processing (see Subsection 3.2), `examples` offering a variety of problem solving scripts serving to exemplify the use of PySAT, `engines` allowing a user to implement external propagators (see Subsection 3.3) as well as optional `pb` and `allies` modules interfacing with third-party libraries. While `pb` provides access to a list of pseudo-Boolean constraint encodings [50] by means of using the PyPBLib library [47, 41], `allies` is meant to offer direct access to external tools developed by the SAT community. Currently, a user can access the ApproxMCv4 approximate model counter [55, 54] and Unigen almost-uniform sampler [12, 11, 54] through the `allies` module. A simplified view on the current architecture of PySAT and its modules interconnection is shown in Figure 1.

3 Selected Novel Functionality

There have been numerous changes made in PySAT since its creation [24]. These include the support for additional SAT solvers, e.g., CaDiCaL 1.0.3, 1.5.3, and 1.9.5 [9] and MiniSat-based [18] solvers Glucose 4.2.1 [4, 22], MapleSAT [28, 31], MapleCM [29], MapleLCMDistChronoBT [40], MergeSat 3.0 [30, 37], CryptoMiniSat [56], additional problem-solving tools like an award-winning MaxSAT solver RC2 [25], smallest minimal unsatisfiable subset (SMUS) extractor OptUx [27], and minimal hitting set enumerator Hitman [13]. As a single paper cannot encompass all the novel changes made, this section briefly overviews three selected components of the framework. (For a complete account of PySAT's capabilities, please refer to the online documentation.)

3.1 Arbitrary Boolean Formulas

In addition to the standard (W)CNF(+) functionality[1] of the original release of PySAT, the current version of the framework offers to users the ability to create *arbitrary* Boolean formulas and, importantly, to clausify them on demand through Tseitin transformation [57]. This may be of special importance to non-SAT researchers who are willing to use SAT for modeling and solving the problems from the scientific domains of their interests. To this end, PySAT exposes atomic expressions and various kinds of logic connectives (all inheriting from a base class `Formula`) as building blocks for creating complex Boolean formulas. For instance, variables can be created as atomic formulas, i.e., objects of the type `Atom`; they can be connected to one another using `And`, `Or`, `Neg`, and `Implies`, among a few other connectives.

▶ **Example 1.** A user may apply the bottom-up formula construction starting from variables, along these lines: `x, y, z = [Atom(c) for c in 'xyz']; f = ~(~x >> y) | (x & z)`, which will create a formula $f \triangleq \neg(\neg x \rightarrow y) \vee (x \wedge z)$.[2]

Furthermore, a user may employ `CNF` objects as components of larger formulas connected to the rest with any of the existing logic operators. To facilitate the use of arbitrary formulas, their on-the-fly clausification, integration of the `CNF` objects, as well as cardinality and pseudo-Boolean constraints handling, the framework offers a simple yet powerful manager of integer variable identifiers referred to as `IDPool`.

▶ **Example 2.** Given formula f in Example 1, calling `f.clausify()` will produce a list of clauses `[[1, 2, -3], [3, -1], [3, -2], [1, -5], [4, -5], [5, -1, -4], [-3, 5]]`, which invokes `IDPool` to automatically assign integer identifiers `1`, `2`, and `4` to variables x, y, and z as well as identifiers `-3` and `5` for the two terms of the disjunction $\neg(\neg x \rightarrow y)$ and $x \wedge z$, respectively.

3.2 CNF Processing

CNF formulas can be (pre-)processed producing *equisatisfiable* CNF formulas using the facilities of PySAT's module `process`. This includes running some of the well-known *preprocessing* techniques such as bounded variable elimination, blocked clause elimination, failed literal probing, among many others [10]. A user may specify how many rounds of CNF processing should be applied as well as select the techniques to apply. In practice, applying such techniques may lead to formulas that are simpler to deal with than the original formulas. Furthermore, the power of formula processing may often suffice for proving unsatisfiability, without the need for a subsequent SAT call. Otherwise, assignments satisfying a processed formula can be mapped back to the original formula. Note that this module is implemented by exposing the preprocessing functionality offered by the CaDiCaL SAT solver [9].

▶ **Example 3.** The status of the result formula produced by the processor can be used to indicate whether or not the processor determined the initial formula to be unsatisfiable. As an example, consider an unsatisfiable input CNF formula $(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1)$. If fed with this formula, the processor returns a new (processed) formula object containing an empty clause and whose status is set to false, signifying unsatisfiability of the original formula, as follows:

[1] CNF and WCNF formulas in PySAT [24] are represented as lists of clauses, each being a list of literals such that a positive integer `i` acts as a literal x_i while the negation $\neg x_i$ is represented by `-i`. CNF+ and WCNF+ formulas can also contain native cardinality constraints *understandable* by solvers like MiniCard [38].

[2] Observe how operators `~`, `&`, `|`, and `>>` are overloaded to act as `Neg`, `And`, `Or`, and `Implies`, respectively.

```
>>> from pysat.process import Processor
>>> proc = Processor(bootstrap_with=[[-1, 2], [-2, 3], [-1, -3], [1]])
>>> processed = proc.process()
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[[]], False # result contains an empty clause and is unsatisfiable
```

▶ **Example 4.** Recall that the `process` module ensures equisatisfiability of an input formula and result formula. Furthermore, a satisfying assignment for the original formula can be restored given a satisfying assignment for the processed one. Consider a CNF formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (x_1)$, which has a single model satisfying literals x_1 and $\neg x_2$. The processor constructs an empty (processed) formula and declares that the original formula is *not* determined to be unsatisfiable. Observe how one can get the unique assignment mentioned above restored from an assignment obtained for the processed formula:

```
>>> from pysat.process import Processor
>>> from pysat.solvers import Solver
>>> proc = Processor(bootstrap_with=[[-1, -2], [1, 2], [1]])
>>> processed = proc.process()
>>> print('{0}, {1}'.format(processed.clauses, processed.status))
[], True # result has no clauses and is not found to be unsatisfiable
>>> with Solver(bootstrap_with=processed) as solver:
...     st, mod = solver.solve(), solver.get_model()
...     print('status: {0}, model: {1}'.format(st, proc.restore(mod)))
status: True, model: [1, -2] # result is confirmed to be satisfiable
                             # and the correct model is restored
```

3.3 External Engines

Recent work [20] proposed an extension to the IPASIR interface [6] referred to as IPASIR-UP and added support for the new interface in CaDiCaL [9]. IPASIR-UP allows a user to specify an external propagator and attach it to a SAT solver supporting the interface. It has shown to be helpful in various practical scenarios when non-clausal reasoning is useful, e.g., in satisfiability modulo theories (SMT) solving [7, 8]. As a result, the interface is deemed highly valuable for extending applicability of SAT, especially if supported by additional solvers.

As handling low-level interaction between a SAT solver and an external propagation engine is a challenging and tedious task, PySAT takes on the mission to provide similar functionality directly in Python. Although using a reasoning engine written in a high-level programming language should arguably be slower than doing the same in a low-level language, this possibility aligns with the aims of PySAT to make the advanced SAT technology easy to use in practice.

The PySAT API for implementing external propagators offered by module `engines` is shown in Listing 1. (User-defined propagators should be defined as inheriting from the class `Propagator`.) Initially, the idea was to expose exactly the same interface as dictated by IPASIR-UP [20]. However, some of the methods in IPASIR-UP inherit the literal-by-literal interface of IPASIR, e.g., clauses are transferred through multiple calls to the same method, each passing a single literal. This overhead becomes noticeable if implemented in Python. Therefore, the interface is slightly modified in PySAT aiming to reduce the overhead of making repeated Python calls from C++ code. First, a clause is handed by the propagator to the solver as a single list of literals, which applies to methods `propagate()`, `provide_reason()`, and `add_clause()`. Internally, the C++ wrapper still follows the original interface by putting the literals into a queue. Second, PySAT gets rid of the *original* `has_external_clause()`

Listing 1 Interface for implementing external propagators in PySAT provided by module `engines`.

```
class Propagator(object):
    def on_assignment(self, lit: int, fixed: bool = False) -> None:
        pass       # receive a new literal assigned by the solver

    def on_new_level(self) -> None:
        pass       # get notified about a new decision level

    def on_backtrack(self, to: int) -> None:
        pass       # process backtracking to a given level

    def check_model(self, model: list[int]) -> bool:
        pass       # check if a given assignment is indeed a model

    def decide(self) -> int:
        return 0   # make a decision and (if any) inform the solver

    def propagate(self) -> list[int]:
        return [] # propagate and return inferred literals (if any)

    def provide_reason(self, lit: int) -> list[int]:
        pass       # explain why a given literal was propagated

    def add_clause(self) -> list[int]:
        return [] # add an(y) external clause to the solver
```

check assuming that no clause is available if `add_clause()` returns `[]`. Finally, to avoid situations when a propagator does not communicate any useful information to the solver and runs for nothing, PySAT allows the propagator to disable itself on the fly (it is up to the author of a propagator to decide when it should happen) such that it will be invoked only when a SAT solver comes up with an assignment to be checked by `check_model()`. On-the-fly re-enabling of the propagator whenever it is favorable is also possible.

4 Distribution

The framework is distributed as an *open-source* project[3] with detailed installation instructions allowing a user to compile all the necessary and/or optional C++ components of the framework and get PySAT ready for use on their local machine. The easiest way to get PySAT is to install a pre-compiled binary wheel, i.e., Python package, from the PyPI repository [48] as follows:

```
$ pip install python-sat
```

Besides the source code distribution, the list of released binary wheels is quite extensive and contains 119 pre-built distributions targeting various versions of Linux, macOS, and Windows operating systems. Finally, PySAT is a part of the Pyodide project [46] whose aim is to supply a version of Python compiled to WebAssembly and deliver a large collection of scientific computing packages available for execution *entirely in a web browser*. A nice

[3] `https://pysathq.github.io`

side effect of this is that a user can implement their SAT-based solutions in Python and/or Javascript, to be operated in a browser, thus potentially expanding practical applicability of the SAT technology without the need to ever install PySAT.

5 Experimenting with External Engines

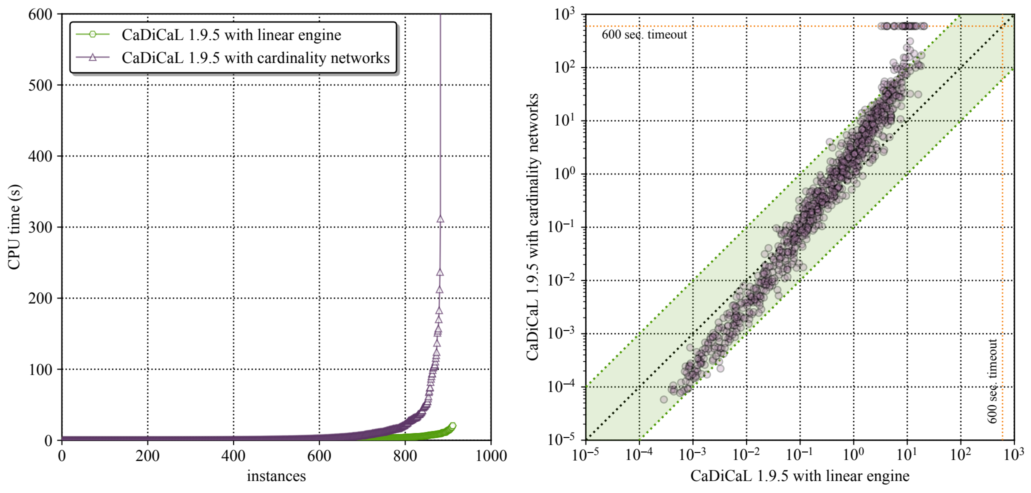
This section aims at showcasing the use of external propagator functionality offered in PySAT given two practical scenarios. We are essentially interested in testing how costly it is to run an external engine alongside CaDiCaL in practice and whether it defeats the purpose of implementing propagators in Python. With this in mind, we implemented an *example* propagator referred to as `BooleanEngine`, which should be general enough to attach various kinds of constraints on Boolean variables, including unweighted and weighted *linear* (i.e., cardinality and pseudo-Boolean, respectively) constraints or XOR-propagators, among other kinds of constraints. We implemented both weighted and unweighted linear constraints for the purpose of the experiment. Hereinafter, `BooleanEngine` reasoning on linear constraints is referred to as the *linear engine*. Here, we would like to remind the reader that the point is not to show that implementing a propagator in Python will necessarily outperform a low-level solution but rather to show that it may pay off in terms of the time invested in the development process, sacrificing little of the overall performance. Both experiments were run on a MacBook Pro running macOS Sonoma 14.3.1 with a 10-core Apple M1 Pro CPU and 32GByte RAM.

▶ Remark 5. The experimental results are presented in the form of cactus and scatter plots, e.g., see Figure 2a and Figure 2b, respectively. A cactus plot depicts multiple lines, each representing a particular competitor in terms of the statistic information on how many instances (on the X-axis) are successfully solved by this competitor within a given time limit (on the Y-axis). A scatter plot depicts in instance-by-instance comparison of two approaches by means of a set of points with coordinates (x, y), each representing a particular problem instance such that coordinate x signifies the time spent by one of the approaches (shown on the X-axis) dealing with this particular instance while coordinate y denotes the time spent on this instance by its competitor (shown on the Y-axis). Note that the green band in the scatter plots denotes the area where no approach outperforms the other by more than an order of magnitude.

5.1 Model Enumeration for Cardinality Constraints

Hereinafter, the first experiment is devoted to running the engine with cardinality constraints. In this case, we randomly generate 1000 systems of (unweighted) linear inequalities over 20 variables, i.e., each such inequality is of the form $\sum_{i=1}^{20} w_i \cdot l_i \leq v$, where $l_i \in \{x_i, \neg x_i\}$ such that $x_i \in \{0, 1\}$ and $w_i \in \{0, 1\}$, $v \in \{0, 1, \ldots, 20\}$. Inconsistent systems are filtered out, which results in 911 remaining problem instances. These remaining instances are either given to CaDiCaL augmented with the linear engine or given to pure CaDiCaL dealing with CNF formulas encoding the linear systems using cardinality networks [3].

Given that a solver may be lucky in finding a single satisfying assignment, the experiment is set to enumerate *all models* of the corresponding formulas. Depending on the formula, the number of models to enumerate varies from 1 to 1,044,905. Model enumeration is done by adding clauses blocking previously found models. For a fair comparison, the performance of a tool is measured as the overall time spent during the enumeration process, thus, ignoring the encoding time. The timeout value set for enumerating the models of a single formula is 10 minutes.



(a) Overall performance. **(b)** Instance-by-instance comparison.

Figure 2 Linear engine vs cardinality networks.

The performance of both competitors, i.e., CaDiCaL with and without the external linear propagator, is depicted in Figure 2. As can be observed, the configuration running the linear engine outperforms the competitor operating on CNF encodings of the inequality systems. In particular, the winner manages to solve all the instances spending *at most* 20.27 seconds per instance while the competitor on average spends much more time per instance and times out on 28 out of 911 instances.

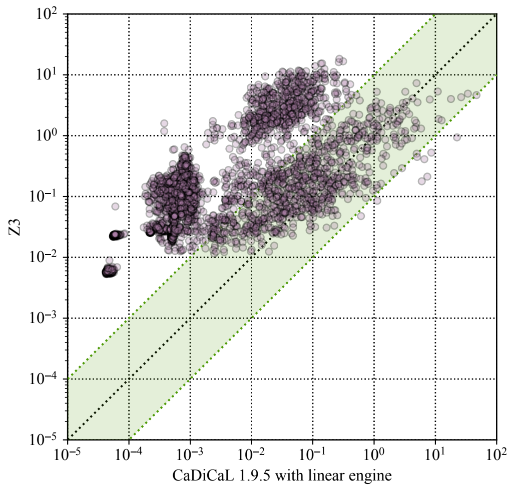
5.2 Computing Formal Explanations for Tree Ensembles

The second experiment considers a more practical setting where we use CaDiCaL augmented with the external linear engine for computing and enumerating formal abductive explanations [52, 26, 33] for tree ensembles trained with the XGBoost algorithm [14]. Without diving into details, the task here is given a machine learning (ML) classification function $\kappa : \mathbb{F} \to \mathcal{K}$ mapping points in feature space $\mathbb{F}$ defined over $n = |\mathcal{F}|$ features $j \in \mathcal{F}$ to a class in $\mathcal{K}$ and a particular prediction $\kappa(\mathbf{v}) = c$, $\mathbf{v} \in \mathbb{F}$, and $c \in \mathcal{K}$, to compute a subset-minimal subset of features $\mathcal{X} \subseteq \mathcal{F}$ such that

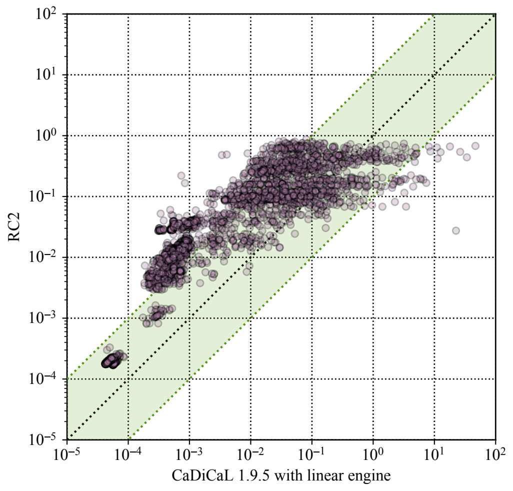
$$\forall(\mathbf{x} \in \mathbb{F}).\left[\bigwedge\nolimits_{j \in \mathcal{X}} (x_j = v_j)\right] \rightarrow (\kappa(\mathbf{x}) = c)$$

Computing such a subset $\mathcal{X}$ requires one to make multiple calls to a reasoning oracle dealing with a logical representation of classifier κ. We reuse the propositional encoding of tree ensemble models proposed in [23] where feature domains and tree paths are CNF-encoded and each node in a tree is represented by a Boolean variable while a class weight $w \in \mathbb{R}$ assigned by a tree's terminal node t is modeled as a weighted soft clause (t, w). However, the use of the linear engine alongside CaDiCaL enables us to model the class selection process of boosted trees directly using pseudo-Boolean constraints, which compare the sums of weights for various classes, rather than by means of a MaxSAT objective function as in [23].

This experiment is twofold. First, it compares the performance of CaDiCaL augmented with the linear engine against the MaxSAT and SMT approaches [23] to computing a single explanation for the XGBoost models trained on a variety of publicly available datasets. (The SMT approach makes use of Z3 solver [16, 21] while the MaxSAT approach employs an



(a) Comparison against Z3.



(b) Comparison against RC2.

Figure 3 CaDiCaL with linear engine vs MaxSAT and SMT on the task of explanation extraction.

optimized version of RC2 [25, 23].) The models trained comprise 50 trees per class, each of depth 1–5; the training (test, resp.) accuracy of these models is above 97% (85%, resp.). The experiment targets explaining 200 randomly selected instances from the corresponding datasets,[4] which resulted in 3755 individual problem instances. Second, it checks scalability of explanation *enumeration* with the MaxSAT vs CaDiCaL with external linear engine,[5] both set to enumerate 100 explanations for each of the models and instances considered above.

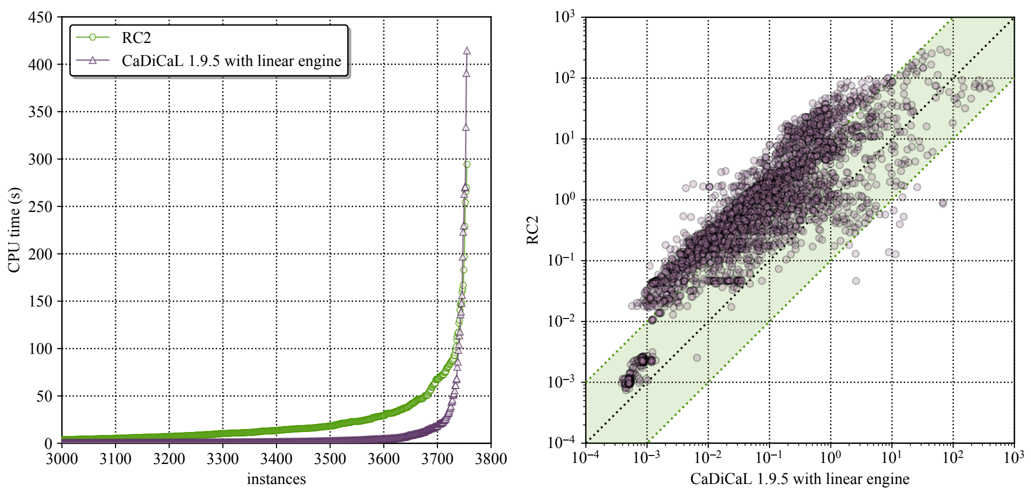
The results of single explanation computation is detailed in Figure 3. Observe that the slowest among the considered approaches is SMT while the fastest overall is MaxSAT. We should also mention that CaDiCaL with the linear engine is significantly less robust than MaxSAT as the time spent to extract an explanation with CaDiCaL varies much more. Finally, the use of the linear engine starts having performance drops if we increase models sizes; no such performance drops occur for MaxSAT or SMT.

The performance in explanation enumeration is detailed in Figure 4. Interestingly, this is where CaDiCaL with the external engine thrives and tends to outperform MaxSAT despite the fact that the engine is called hundreds of thousands of times per problem instance, which may be seen as surprising.

6 Related Work

While PySAT was originally inspired by PySMT's [21] capability of interfacing with various SMT solvers, there are Python APIs targeting *individual* SAT solvers, e.g., *PyMiniSolvers* [45] providing an API to MiniSat and MiniCard, *pycosat* [43] with the Python interface to PicoSAT, *satispy* [51] offering an API for MiniSat and lingeling, *pylgl* [44] for working with lingeling, and *pycryptosat* providing access to CryptoMiniSat [56, 53, 15]. Another framework called *OptiLog* [2, 1] offers a unified interface to multiple SAT solvers as well as access to cardinality

[4] If a dataset has fewer than 200 instances, we explain each of the $n < 200$ *available* instances.
[5] The implementation of [23] does not support explanation enumeration with SMT, which is why we do not compare against SMT in the explanation enumeration mode.



(a) Overall performance. **(b)** Instance-by-instance comparison.

Figure 4 MaxSAT vs CaDiCaL with linear engine on the task of explanation enumeration.

and pseudo-Boolean constraint encodings. However, while OptiLog provides a user with a unified interface *iSAT* (through C++ and Python) to attach a SAT solver of their interest as well as with a way to configure, fine-tune, and benchmark it, PySAT's goal is different. Namely, PySAT is a fully open-source Python framework aiming to simplify prototyping with SAT oracles, even for researchers with little experience with SAT solving. It delivers a large range of solvers pre-installed accessible through the same API as well as a wealth of facilities to manipulate Boolean formulas and implement user-defined constraint reasoners.

7 Conclusions

It has been a number of years since the original release of the PySAT framework [24]. This paper provides a brief overview of its current capabilities. Besides a larger number of SAT solvers and problem-solving scripts integrated into the toolkit, these capabilities include facilities to manipulate arbitrary Boolean formulas and linear constraints as well as external reasoning engines by exploiting IPASIR-UP [20], and formula (pre-)processing and clausification [57]. PySAT is an easy-to-deploy and fully open-source Python package, whose mission is to make the advances of SAT universally accessible for solving problems arising in a wide range of scientific domains. Hopefully, the broader SAT community will assist this by contributing to PySAT with additional features and with proposals for further improvements.

References

1 Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. OptiLog V2: model, solve, tune and run. In *SAT*, pages 25:1–25:16, 2022.

2 Carlos Ansótegui, Jesus Ojeda, António Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. OptiLog: A framework for SAT-based systems. In *SAT*, pages 1–10, 2021.

3 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *SAT*, pages 167–180, 2009.

4 Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving Glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT*, pages 309–317, 2013.

5 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.

6 Tomás Balyo, Armin Biere, Markus Iser, and Carsten Sinz. SAT race 2015. *Artif. Intell.*, 241:45–65, 2016.

7 Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 1267–1329. IOS Press, 2021.

8 Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

9 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53, 2020.

10 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In *Handbook of Satisfiability*, pages 391–435. IOS Press, 2021.

11 Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. On parallel scalable uniform SAT witness generation. In *TACAS*, pages 304–319, 2015.

12 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Balancing scalability and uniformity in SAT witness generator. In *DAC*, pages 60:1–60:6, 2014.

13 Karthekeyan Chandrasekaran, Richard M. Karp, Erick Moreno-Centeno, and Santosh S. Vempala. Algorithms for implicit hitting set problems. In *SODA*, pages 614–629, 2011.

14 Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

15 CryptoMiniSat. `https://github.com/msoos/cryptominisat/`.

16 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

17 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75, 2005.

18 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

19 Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

20 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: user propagators for CDCL. In *SAT*, pages 8:1–8:13, 2023.

21 Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop*, 2015.

22 Glucose 3, Glucose 4.1, and Glucose 4.2.1. `http://www.labri.fr/perso/lsimon/glucose/`.

23 Alexey Ignatiev, Yacine Izza, Peter J. Stuckey, and João Marques-Silva. Using MaxSAT for efficient explanations of tree ensembles. In *AAAI*, pages 3776–3785, 2022.

24 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

25 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019.

26 Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. Abduction-based explanations for machine learning models. In *AAAI*, pages 1511–1519, 2019. `doi:10.1609/AAAI.V33I01.33011511`.

27 Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and Joao Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP*, pages 173–182, 2015.

28 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *SAT*, pages 123–140, 2016.

29 Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü. An effective learnt clause minimization approach for CDCL SAT solvers. In *IJCAI*, pages 703–711, 2017.

30 Norbert Manthey. The MergeSat solver. In *SAT*, pages 387–398, 2021.

31 MapleSAT. `https://maplesat.github.io/`.

32 Joao Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits*. PhD thesis, University of Michigan, 1995.

33 João Marques-Silva and Alexey Ignatiev. Delivering trustworthy AI through formal XAI. In *AAAI*, pages 12342–12350, 2022.

34 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, pages 133–182. IOS Press, 2021.

35 Joao Marques-Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

36 Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

37 MergeSat. `https://github.com/conp-solutions/mergesat`.

38 MiniCard 1.2. `https://github.com/liffiton/minicard/`.

39 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

40 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT*, pages 111–121, 2018.

41 Tobias Philipp and Peter Steinke. PBLib - A library for encoding pseudo-Boolean constraints into CNF. In *SAT*, pages 9–16, 2015.

42 Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

43 pycosat. `https://github.com/conda/pycosat`.

44 pylgl. `https://github.com/abfeldman/pylgl/`.

45 PyMiniSolvers. `https://github.com/liffiton/PyMiniSolvers/`.

46 Pyodide. `https://pyodide.org/`.

47 PyPBLib. `https://pypi.org/project/pypblib/`.

48 PyPI. `https://pypi.python.org/`.

49 PyPI 2FA Security Key Giveaway. `https://pypi.org/security-key-giveaway/`.

50 Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 1087–1129. IOS Press, 2021.

51 satispy. `https://github.com/netom/satispy/`.

52 Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining Bayesian network classifiers. In *IJCAI*, pages 5103–5111, 2018. `doi:10.24963/IJCAI.2018/708`.

53 Mate Soos. Enhanced gaussian elimination in dpll-based SAT solvers. In *POS@SAT*, pages 2–14, 2010.

54 Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *CAV*, pages 463–484, 2020.

55 Mate Soos and Kuldeep S. Meel. BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *AAAI*, pages 1592–1599, 2019.

56 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT*, pages 244–257, 2009.

57 G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

58 Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Parallel Clause Sharing Strategy Based on Graph Structure of SAT Problem

Yoichiro Iida
Graduate School of Information Science and Technology, The University of Tokyo, Japan

Tomohiro Sonobe
National Institute of Informatics, Tokyo, Japan

Mary Inaba
Graduate School of Information Science and Technology, The University of Tokyo, Japan

Abstract

Parallelization of SAT solvers is an important technique for improving solver performance. The selection of the learnt clauses to share among parallel workers is crucial for its efficiency. Literal block distance (LBD) is often used to evaluate the quality of clauses to select. We propose a new method, *Parallel Clause sharing based on graph Structure* (PaCS), to select good clauses for sharing. First, we conducted three preliminary experiments to assess the performance of LBD in parallel clause sharing: a performance comparison between the LBD and clause size, an analysis of the utilization of shared clauses, and a comparison of the LBD values of shared clauses at originating and receiving workers. These experiments indicate that the LBD may not be optimal for learnt clause sharing. We attribute the results to the LBD's inherent dependency on decision trees. Each parallel worker has a unique decision tree; thus, a sharing clause that is good for its originating worker may not be good for others. Therefore, we propose PaCS, a search-independent method that uses the graph structure derived from the input CNF of SAT problems. PaCS evaluates clauses using their edges' weight in the variable incidence graph. Using the input CNF's graph is effective for parallel clause sharing because it is the common input for all parallel workers. Furthermore, using edge weight can select clauses whose variables' Boolean values are more likely to be determined. Performance evaluation experiments demonstrate that our strategy outperforms LBD by 4% in the number of solved instances and by 12% in PAR-2. This study opens avenues for further improvements in parallel-solving strategies using the structure of SAT problems and reinterpretations of the quality of learnt clauses.

2012 ACM Subject Classification Computing methodologies → Heuristic function construction; Mathematics of computing → Solvers

Keywords and phrases SAT Solver, Structure of SAT, Parallel application, Clause Learning

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.17

Acknowledgements We would like to express our sincere gratitude to the reviewers for their valuable comments and constructive feedback, which have greatly improved the quality of this paper. Additionally, we are grateful to the members of the Imai Laboratory for their insightful discussions and support throughout this research.

1 Introduction

Satisfiability (SAT) solvers are tools that determine whether the input Boolean formula is satisfiable or not. Conflict-driven clause learning (CDCL) SAT solvers [28, 29] are widely used because of their high efficacy in many industrial SAT problems. Clause learning [8] is an important component of CDCL solvers. Clause learning generates new clauses (learnt clauses) to prevent the solver from repeating the wrong assignments. Learnt clauses significantly improve search efficiency by pruning the search space. Parallelization of SAT solvers is also an important technique for improving solver performance. The information can be shared between parallel workers to enhance the overall parallel efficiency. In parallel SAT solvers, workers exchange their acquired learnt clauses. However, because the solver often learns over millions of clauses during its search, it is impractical to share all of them. Therefore, literal block distance (LBD) [4] is often used for clause selection. LBD was originally proposed as an evaluation metric of clause quality. LBD has been used by many successful solvers in SAT competitions [1], in both sequential and parallel solvers.

First, in this study, to assess the performance of the LBD in parallel clause sharing, we conducted three preliminary experiments: (1) a comparison of solver performance using LBD and clause size, (2) an analysis of the utilization ratio of shared clauses, (3) a comparison of LBD values of shared clauses between originating and receiving workers. The results of the experiments indicate that LBD may not be the optimal metric for learnt clause sharing; the size of the clause performs competitively with (actually slightly better than) LBD, and the utilization ratio of the shared clause is low. We attribute the results to the LBD's inherent dependency on the search state. The LBD value indicates the number of variable decision levels in the clause. The decision level corresponds to the depth of the branching decision tree. Several factors influence the tree, including the decision strategy, learned clauses, heuristics status, and solver configurations. We call these varying conditions "search states" and say LBD has an inherent dependency on the search state. However, each parallel worker has a unique search state; particularly in portfolio-type parallel SAT solvers, they should be different. Therefore, a sharing clause good for its originating worker, as measured by the LBD, may not be good for others. According to Audemard and Simon, "*LBD is relative to the current search of each solver, and a good clause for one thread may not be good for another one.*" [5].

Therefore, we propose *Parallel Clause sharing based on graph Structure* (PaCS), a search-independent method that uses the graph structure derived from the input CNF of SAT problems. It evaluates clauses using a metric [21], that measures the strength of the connection between variables in the clause and their neighboring variables. The weight of the edge in the variable incidence graph (VIG) quantifies the strength of variables connection. This metric was originally proposed for the deletion of learnt clauses and demonstrated comparable performance to LBD. We assume that using the graph of input CNF is effective for parallel clause sharing because it is a common input for all parallel workers, enabling it to identify good clauses for all workers. Furthermore, using edge weight can select clauses whose variables' Boolean values are more likely to be determined. A heavy edge implies that the pair of variables is contained in a short clause or many clauses. Thus, variables with heavier edges have a better chance of undergoing propagation. Moreover, various studies have shown a relationship between the quality of clauses and their graph structures [20, 35].

We implemented a parallel solver that uses our proposal clause sharing selection method and conducted performance evaluation experiments to compare it with LBD. The contributions of our research are as follows:

1. Through three preliminary experiments, we have demonstrated that LBD may not be the optimal solution for selecting learnt clauses to share among parallel workers.
2. We propose PaCS, which uses a metric that represents the strength of the connection in the graph structure for parallel clause sharing.
3. We compared the performance of a parallel solver using PaCS with LBD and found that it outperforms solvers using LBD by 4% in the number of solved instances and by 12% in PAR-2.

[1] SAT competition `http://www.satcompetition.org/`

The remainder of this paper is organized as follows: Section 2 introduces SAT solvers and their techniques, including learnt clauses and the concept of the graph structure of SAT problems. Section 3 discusses related work. Section 4 presents our empirical observations of LBD for clause sharing. Section 5 defines the proposed method for parallel clause sharing using PaCS. Section 6 presents the results of the PaCS performance evaluation. Finally, Section 7 summarizes the study and suggests future research directions.

2 Preliminaries

2.1 Satisfiability problem and SAT solver

The SAT problem determines whether at least one Boolean variable assignment can satisfy a given logical formula. If an assignment can satisfy all clauses, the formula is satisfiable; otherwise, it is unsatisfiable (UNSAT). The formula is generally provided in conjunctive normal form (CNF), wherein variables are combined into clauses with disjunctions, and the clauses are combined with conjunctions. A formula is in CNF if it possesses the form $(C_1 \wedge C_2 \wedge \cdots \wedge C_m)$, where C_i represents a clause. Each clause is a disjunction of literals $(L_{i,1} \vee L_{i,2} \vee \cdots \vee L_{i,n})$, where $L_{i,j}$ represents a literal in clause C_i. A literal is a variable or its negation. An example of a CNF formula is $(x \vee y) \wedge (\neg y \vee z)$. Here, $(x \vee y)$ and $(\neg y \vee z)$ are clauses, and x, y, $\neg y$, and z are literals where y and $\neg y$ represent the positive and negative forms of variable y, respectively.

The SAT solvers are programs for solving SAT problems. It is used to solve real-world problems encoded in SAT, such as computer-aided proof [18] and binary neural network verification [11]. Davis–Putnam–Logemann–Loveland (DPLL) algorithm [13] is the basis of most SAT solvers. The DPLL algorithm contains decisions – providing an assumption of Boolean (True or False) value to a variable – and incorporates propagation – determining other variables' Boolean values as the logical consequences of the decision. A conflict occurs when a clause becomes false due to a wrong decision. Then, the previous decisions are canceled (called backtrack), and the solver makes another decision. Modern SAT solvers are conflict-driven clause-learning (CDCL) solvers [28, 29] that are based on the DPLL algorithm. The CDCL solvers incorporate various techniques and heuristics [15, 27, 29], making it possible to solve large and complex SAT problems efficiently. Probabilistic algorithms [34] are another approach that randomly changes the Boolean values of variables until a solution is found. In this paper, we focus on the improvement of CDCL SAT solvers.

2.2 Clause Learning

Clause learning [8] is an essential technique for CDCL solvers, aiming at improving efficiency by pruning the search space. When a solver finds a conflict, it analyzes the root cause and derives the counter-example as a new clause (learnt clause) to avoid revisiting similar unsatisfiable assignments in future searches. Modern solvers often learn over millions of clauses during searches. However, it is difficult to maintain all of the clauses due to the cost of checking these clauses during propagation. Clause evaluation is necessary to selectively retain more valuable clauses for future searches. The clause size, literal block distance (LBD) [4], and activity (or clause version of a variable state independent decaying sum, cVSIDS) [2] are widely adopted metrics. The size of a clause represents the number of literals contained in

[2] MiniSat `http://minisat.se/`

the clause. Shorter clauses are more effective in reducing the search space. Given a clause c and literals l, $size(c) := |\{l \in c\}|$. The LBD [4] is a popular clause evaluation metric and is widely adopted by many state-of-the-art SAT solvers. The LBD value of a clause is defined as the number of different decision levels to which the literals in the clause belong. The decision level indicates the depth of the branching decision tree. The LBD value of a clause c can be calculated as $LBD(c) := |\{d(l) : l \in c\}|$ where $d(l)$ represents the decision level of literal l. A clause with a lower LBD is considered more valuable. In addition to clause evaluation, the LBD is used in other heuristics such as restart strategy [9] and decision branching [12]. Activity (or cVSIDS) measures how frequently a clause is used in the conflict analysis. The more used clauses are more valuable, and a higher score is provided for recently used ones. Among these three, the results of evaluation by LBD and Activity are based on the current search state, such as the decision tree and maintained learnt clauses. Therefore, these values are different at different search states, while size shows constant values.

2.3 Parallel SAT solver

Parallelization of SAT solvers is an important technique for improving solver performance, leveraging plentiful computing resources such as affordable multi-core processors and cloud computing services. A parallel SAT solver simultaneously utilizes multiple cores to solve SAT problems. It shares valuable learnt clauses among workers to enhance parallel efficiency. This may potentially help to solve larger or more complex problems more efficiently. Two main approaches are adopted in parallel solvers: divide and conquer [10] and portfolio [19]. The divide and conquer approach splits the problem into smaller sub-problems, solves them independently, and then combines the solutions. This approach possesses excellent scalability because assigning Boolean values to some variables can easily split the search space. However, load balancing and UNSAT proof are the challenges of the approach. The portfolio approach, on the other hand, does not split the problem; it uses multiple different strategies (i.e., search) simultaneously for the same problem, and it subsequently selects the best solution among these strategies. Although this may help resolve the issue of divide and conquer by design, scalability is a challenge due to the difficulty of ensuring various search strategies. In the recent SAT competition, most successful parallel solvers adopted the portfolio approach.

Sharing information between parallel workers is crucial to enhancing overall efficiency. This is usually done by sharing learnt clauses among parallel workers. As each worker explores different search spaces, they generate different learnt clauses. Sharing these clauses may help other workers leverage collective knowledge and avoid redundant searches. However, sharing many clauses increases communication overhead and propagation costs. Therefore, it is essential to appropriately manage the volume and frequency of sharing. With regard to these factors, the size and LBD values of clauses are often used as the selection criteria; the clauses expected to be valuable for other workers are shared based on the evaluation of these metrics. For example, ParKissat-RS [3] and PRS [4], which are the winner in the parallel track of SAT competition 2022 and 2023, respectively, share the clauses with LBD values of one or two by default. Section 3.1 gives a detailed explanation of the method.

[3] ParKissat-RS `https://github.com/shaowei-cai-group/ParKissat-RS`
[4] PRS `https://github.com/shaowei-cai-group/PRS-sc23`

2.4 Structure of SAT

Industrial SAT problems encoded from real-world problems exhibit unique structures and patterns, which can differentiate them from randomly generated problems. For example, the treewidth of the graph representation of the industrial SAT problem is notably small [14]. The centrality of SAT refers to how central or critical a variable or clause is within the problem's structure [24]. Industrial problems exhibit a clear community structure [1, 31], and some correlation between the degree of the community (modularity) and runtime of the solver is known [30]. Additionally, many structural properties, such as mergeability [37] and the role of backdoor variables [36], have been studied to understand the efficiency of SAT solvers.

The graph of the SAT problem is often represented using a variable incidence graph (VIG). In this graph, nodes represent variables, and edges indicate the existence of clauses containing these variables. Each edge has a weight, and it is determined based on the strength of the variable connection. Heavier weight is given to edges whose variables coexist in more or shorter clauses. The definition of weight $w(e_{v_i,v_j})$ of the edge between v_i and v_j is $\sum_{c \in C, v_i, v_j \in c} 1/\binom{|c|}{2}$ following the existing research [1]. A clause variable incidence graph (CVIG) is a bipartite graph where variables and clauses are represented as nodes on each side, respectively, and an edge exists between a clause and a variable if the variable exists in the clause. In this study, we use VIG as the graph representation of the SAT problem.

3 Related work

3.1 Clause sharing between parallel workers

ManySAT [17], a portfolio-type parallel SAT solver, uses size as a metric for sharing learnt clauses, where all clauses of size eight or less are shared. Then, they suggested dynamically varying the criterion of the shared clauses [16]. Painless [26] is a parallelization framework that has been recently used by many solvers. This involves running a process called Sharer for clause sharing. The winners of the parallel track SAT competition 2022 and 2023, ParKissat-RS and PRS, adopted the painless framework. They used LBD as the criteria to determine the clauses to be shared. P-KISSAT [6], also based on the painless framework, dynamically varies the criterion for sharing learnt clauses depending on the number of clauses generated by the producer of clauses. Hordesat [7], and its derivative Mallob [33] are popular in large-scale parallel environments (such as high-performance computing) In particular, Mallob achieved excellent results in both the parallel track of the SAT competition 2023 and the cloud track. Hordesat is a large-scale parallel search solver designed for distributed memory and compute node environments. In Hordesat's learnt clause sharing, size is used as a metric for all-worker-to-all clause sharing. Workers add their learnt clauses to a shared buffer. The added clauses are sorted in ascending order of size, and the smaller ones are shared with other parallel workers up to a certain number of clauses. Mallob merged buffers according to the job tree of parallel workers to address problems related to this buffer-sharing system in Hordesat, such as duplicate clauses and the sharing of blank spaces within the buffer.

Several studies focused on the mechanism of sharing learnt clauses rather than just the criterion to share. The ppfolio [5] used an extreme strategy in which no learnt clauses were shared and achieved excellent results in the 2011 SAT competition. Lazaar et al. [25]

[5] ppfolio `https://www.cril.univ-artois.fr/~roussel/ppfolio/`

proposed a sharing strategy that focuses on determining the workers to receive them rather than on the selection of clauses. Audemard et al. [3] proposed psm, which measures the usefulness of a clause in the current search context using variable assignments. Audemard et al. [2] also proposed the psm-based "freeze and activate" strategy, which shares clauses but freezes some that are deemed unnecessary in the current search. While most SAT solvers operate on CPU, there have been efforts to use GPU for rapid and parallel evaluation of clauses [32].

3.2 Clause quality evaluation using graph structure of SAT

Although LBD is more popular than size, size is also used as a secondary metric when two clauses have identical LBD values. Size is a structural property of the SAT graph, which is static in any search state; however, it is considered a poorer metric than LBD. However, Jabbour et al. [22] refocused on the effectiveness of the size metric against LBD. They showed that size-based evaluation with some randomness can improve the solver's performance. Vallade et al. [35] demonstrated the relationship in a clause between the LBD value and the number of communities in the graph representation of SAT. They proposed a novel clause evaluation method that combines the number of communities and the clause's LBD value. Jamali et al. [23] proposed using the structural properties of the SAT problem for heuristics, such as decision and clause evaluation. They selected the betweenness centrality of the variables in a learnt clause as an evaluation metric and demonstrated that the method could improve the solver's performance. We proposed a clause evaluation method using the graph structure derived from the input CNF of the SAT problem, called WANCE [21]. It favorably evaluated clauses with variables that had strong (heavy) edges to their neighboring variables in VIG. Details of the definition are presented in Section 5.

4 Observations

This chapter shows the results of preliminary experiments to assess the performance of the popular method, LBD, in parallel clause sharing. Remember that the LBD value indicates the number of literal blocks in the clause; thus, it depends on the search states. This allows evaluation of the clause's quality optimized according to the current search state. However, in the parallel clause-sharing situation, this can be a disadvantage because the search state of each parallel worker could be different; particularly in the portfolio approach, it should be different in terms of efficiency. To test this hypothesis, we conducted the following experiments.

4.1 Performance comparison of criteria for sharing clause selection

First, we compared the performance difference due to the clause-sharing criteria in a parallel SAT solver. We selected ParKissat-RS, the state-of-the-art and champion parallel solver, in the SAT competition 2022 parallel track. ParKissat-RS decides the clauses to share among parallel workers, as follows. It adopts worker-sharer architecture. Worker generates learnt clauses through search. Only learnt clauses with LBD values of either 1 or 2 are submitted to the buffer. Next, the sharer broadcasts these clauses in the buffer to the other workers up to 1500 literal lengths per sharing cycle. In this experiment, we modified the submission criteria. Size-based criteria submit all clauses with their size of or less than x, where we set x as $1, 2, 3, 5, 8, 10$. Also, no sharing policy ($x = 0$) is compared with LBD. The experimental setup is as follows: We configured the number of parallel workers at 16, set the time limit to 5000 s, and established a memory limit of 128 GB, all by specifying the options in ParKissat-RS. For other settings, we adhered to the default configurations and implementations, including the constraint of sharing literal lengths up to 1500. The benchmark is 400 instances from the SAT Competition 2023. The experiments were conducted on a computer equipped with an AMD Threadripper Pro 3995WX processor with 64 cores and 512 GB RAM (four 128 GB DDR4-3200 MHz slots).

Table 1 Performance comparison among clause sharing criteria using LBD, no sharing, and size. Numbers in the table represent the count of instances in each satisfiability, where SAT denotes instances identified as satisfiable and UNSAT as unsatisfiable within the time limit, respectively.

Criterion	SAT	UNSAT	SAT+UNSAT	PAR-2
LBD (≤ 2)	124	141	265	3715
No sharing	126	131	257	3973
Size (≤ 1)	126	**147**	**273**	3631
Size (≤ 2)	126	141	267	3650
Size (≤ 3)	125	143	268	3619
Size (≤ 5)	125	144	269	3601
Size (≤ 8)	**129**	141	270	**3528**
Size (≤ 10)	121	124	245	4300

The experimental results are shown in Table 1. The first column shows the solvers with each sharing criteria. The numbers in the table represent the number of instances in each satisfiability, where SAT denotes instances identified as satisfiable and UNSAT as unsatisfiable. This experiment showed no significant difference in the performance between LBD and Size-based criterion from 1 to 8 (actually, the performance of size is better than that of LBD). The largest number of solved problems is by the size ≤ 1 criterion, and the lowest PAR-2 score is by the size ≤ 8 criterion. LBD is generally an efficient metric for evaluating learnt clauses compared to their size. However, the results of this experiment suggest that in the task of sharing learnt clauses among parallel workers, LBD performs similarly to size.

4.2 Analysis of utilization differences between imported and learned clauses with same LBD value

Next, we compared the usefulness of shared clauses between "*learned*" and "*imported.*" A clause is termed "*learned*" when a worker derives it during their search. When this clause is shared with other workers (i.e., exported), it becomes "*imported*" for those other workers. Therefore, the same clause will have a different term depending on the worker by which it was acquired. Additionally, the worker that originally learned the clause is referred to as the "*learned*" worker, while the worker that received the shared clause is called the "*imported*" worker. In general, each worker shares the learnt clauses that are expected to be useful to other workers. However, the LBD value can be different in each parallel worker because each worker has a unique search state. In the ParKissat-RS implementation, the LBD value is converted to the clause's size when sharing. Then, it is updated (re-evaluated) according to the search state of the *imported* worker when necessary. This subsection's experiments aim to analyze the difference in the degree of usefulness between *learned* and *imported* clauses. This experiment used the notion of *used%* as below.

4.2.1 Definition of *used%*

The utilization of the clause means that they were used in conflict analysis. We defined *used* as a Boolean value of the clause utilization, which is 1 if the clause is used at least once in conflict analysis and 0 otherwise.

$$\text{used}(c) := \begin{cases} 1 & \text{if } c \text{ is referred in conflict analysis} \\ 0 & \text{otherwise} \end{cases}$$

In the context of this study, the following terms are defined:

- C: the set of all learnt clauses in all parallel workers.
- i: a worker $i \in W$ where W represents all parallel workers.
- ${}_iC$: the set of learned clauses acquired at worker i, $\forall_i C \subset C$.
- ${}_iC^{\text{learn}}$: the set of clauses *learned* by the worker i.
- ${}_iC^{\text{import}}$: the set of clauses *imported* from another worker to i.
- ${}_iC_x$: the set of learned clauses for which LBD value is x at worker i.
- ${}_iC_{(x,y)}$: the set of learned clauses for which LBD value is x and the clause size is y.

For example, ${}_iC_2^{\text{learn}}$ refers to the set of all clauses with LBD of 2 in worker i that are learned at worker i. The LBD value of *imported* clauses C^{import} refers to the value at the *learned* worker before sharing, not the re-evaluated value at the *imported* worker after sharing. Furthermore, the percentage of used clauses within a set C is defined as:

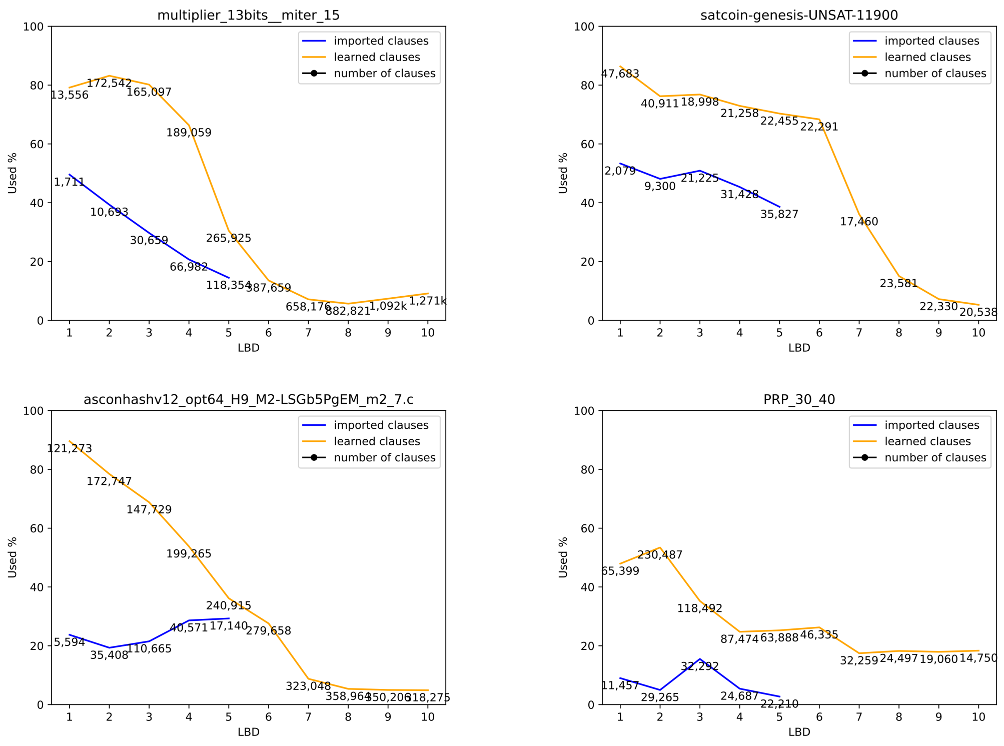
$$\text{used\%}(C) := \frac{\sum_{i \in W, c \in {}_iC} \text{used}(c)}{|\bigcup_{i=1}^{N} {}_iC|}$$

where $|C|$ represents the number of clauses in C.

We used ParKissat-RS as the base solver, with two parallel workers, a time limit of 1000 s, and a benchmark of 400 instances on the SAT competition 2023 benchmark. All learnt clauses and their information were written to an external file, then *used%* was calculated later. The clause was written when the clause was learned, used, deleted, its LBD value was updated, and the search ended. We exclude clauses with a size equal to or less than 2 because they are always watched for propagation and don't have the concept of being used in ParKissat-RS. In its default setting, only the clauses of LBD value of less than or equal to two are shared up to 1500 literal lengths. To analyze the utilization trends of *imported* clauses using broader sets of clauses, we increased the LBD value criterion for clause sharing from the default of 2 to 5, and we expanded the maximum shared literal size from the default of 1,500 to 30,000. We performed experiments on the same computer as the previous experiment.

4.2.2 Analysis of used% according to LBD values

We analyzed the differences of *used%* between *imported* and *learned* clauses for the same LBD value. We specifically compared the used%(C_x^{import}) and used%(C_x^{learn}), respectively, for LBD values x ranging from 1 to 10. Remember that the LBD value of the *imported* clause represents the LBD value being evaluated by the *learned* worker, not by *imported* worker that was reevaluated after sharing. This setup is to observe the gap of use of the same evaluation clauses – given a clause, is there any difference of use % at *learned* worker and *imported* worker? There is no difference if the evaluation shows a common usefulness for all parallel workers. Figure 1 shows the result of randomly selected four instances from all benchmark instances for further analysis. The horizontal and vertical axes present the value of the clause's LBD and the *used%* of these clauses, respectively. The yellow and blue lines represent *learned* and *imported* clauses, respectively. The numbers in the line chart are the number of clauses for each LBD value.



■ **Figure 1** Differences of *used%* at each LBD value between learned and imported clauses.

The yellow line consistently showed that the *used%* decreased as the LBD value increased (i.e., worse clauses). In contrast, for the blue lines, the variation in *used%* with LBD value was less. Furthermore, *used%* were much lower than those of *learned* clauses for the same LBD value. This result indicates that a clause considered useful at an *learned* worker is not necessarily useful for *imported* worker.

4.2.3 Analysis of used% according to LBD values and size

We plotted the data into a heatmap to better understand the relationship between LBD, size, and *used%*. The $used\%(C^{imported}_{(x,y)})$ and $used\%(C^{learned}_{(x,y)})$ of LBD of x and size of y clauses are shown in Figure 2. Two instances out of the previous four are shown as examples owing to the space limitation. The vertical and horizontal axes indicate the LBD value and size, respectively. The cell color indicates the *used%*, where the closer the color is to dark blue, the higher the *used%*, whereas the lighter the color, the lower the *used%*. Same to the previous experiment, the LBD value of the *imported* clause is that at the *learned* worker, not after sharing and at *imported* worker. In the *learned* heatmap, higher *used%* was observed for lower LBD values, and vice versa. The result indicates that the LBD value determines how high or low the *used%* is, regardless of the size of the clause. In contrast, in the *imported* heatmap, the *used%* seems to depend on the size rather than the LBD value. This suggests that the LBD value does not primarily indicate the usefulness of *imported* clauses, as deduced from the *used%* metric. Instead, clause size seems to be associated with *used%*. This result is consistent with that of the previous experiment. The dark blue cells whose LBD and size are equal in the *imported* figure showed considerably high *used%*. However, the number of these clauses was limited.

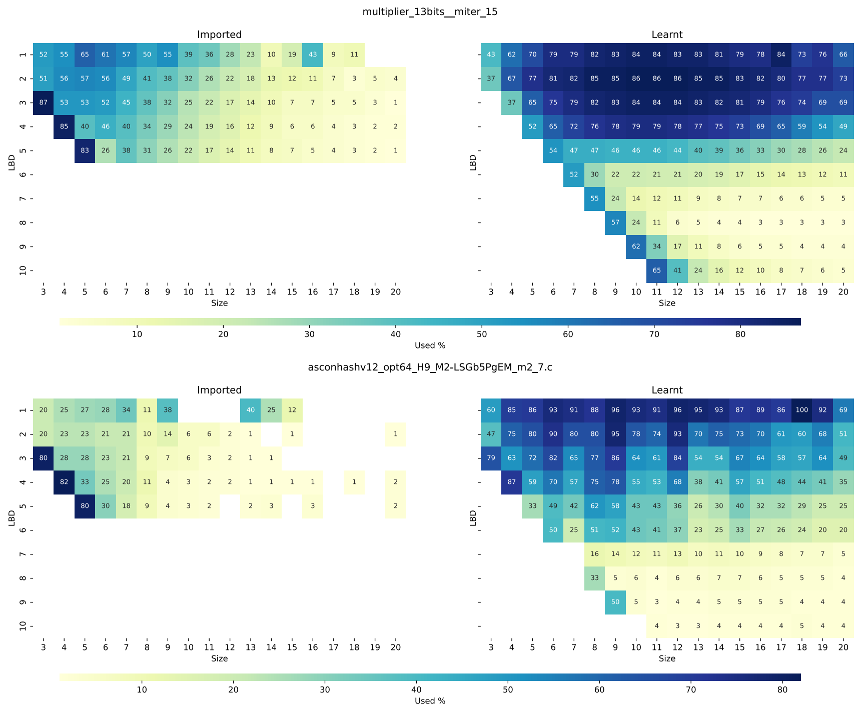


Figure 2 *Used%* heatmap of LBD and size, comparison of clauses learned and imported.

4.2.4 Distribution of used% on LBD among all instances

Figure 3 shows the statistical summary of all the benchmark instances as the box plot. The box plot illustrates the distribution of *used%* among instances. The mean *used%* for each LBD value was used as the value for an instance. The center line of the box represents the median instance, and the ends of the box represent the first and third quartile points, respectively. The edge lines represent the maximum and minimum values, respectively. We excluded outliers from the figure for readability; instances that are 1.5 times smaller or larger than the first and third quartile values. In addition, instances that generate no imported clauses (e.g., the solver finds a solution before no sharing is conducted) are excluded from the result. This figure substantiates that the trend observed in the example instances represents the general trend across many instances.

4.3 Comparison of LBD values of shared clauses at learned and imported workers

In the previous experiments, we compared the *used%* in *learned* worker and *imported* worker, using the LBD value that is calculated at *learned* worker. Next, we quantify the changes in LBD values after sharing; between the values at originating (*learned*) worker and at

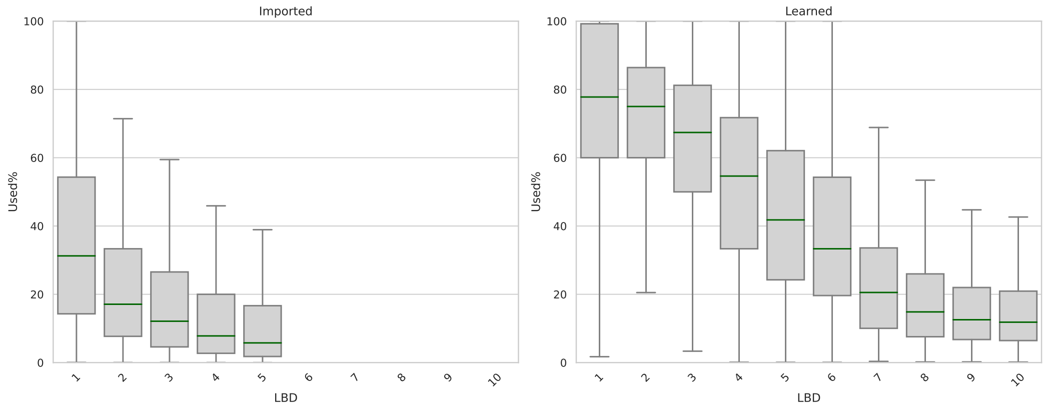


Figure 3 Distribution of *used%* for 400 benchmark instances at each LBD value.

receiving (*imported*) workers after reevaluation. We refer to the LBD value at *learned* worker as *learned* LBD, and that at *imported* worker as *imported* LBD, respectively. The *learned* LBD and *imported* LBD can be different because each worker uses a different decision tree. We compared the change from *learned* LBD to *imported* LBD. This experiment aims at investigating the extent to which the originally highly evaluated (and thus shared) clause is evaluated in the *imported* worker after sharing. This experiment used the same setup as those in the previous experiment. Table 2 shows the result. Each row represents the LBD at the *learned* worker, and each column represents the *imported* LBD. The number in each cell refers to the percentage of clauses defined by

$$\frac{|C_{\text{learned}=x,\text{imported}=y}|}{|C_{\text{learned}=x}|}$$

where x is the *learned* LBD, and y is the *imported* LBD. This indicates the ratio of clauses C_x (LBD $= x$) at *learned* worker that are re-evaluated to y at the *imported* worker. The value is averaged for all instances. Remember that we excluded clauses with a size of one or two following the previous experimental conditions.

Table 2 Comparison of LBD values of shared clauses at originating and receiving workers. 10+ denotes the sum of the percentages for clauses whose size is greater than or equal to 10. The sum of the horizontal axis (rows) adds up to 100%.

		imported LBD									
		1	2	3	4	5	6	7	8	9	10+
learned LBD	1	5%	8%	**33%**	16%	11%	7%	5%	3%	3%	2%
	2	3%	6%	**36%**	14%	9%	6%	5%	4%	3%	3%
	3	3%	5%	5%	**25%**	13%	9%	8%	6%	5%	4%
	4	2%	2%	2%	3%	**18%**	10%	10%	8%	7%	5%
	5	2%	2%	2%	2%	2%	**13%**	10%	9%	9%	8%

If LBD can measure the common quality for all parallel workers, *imported* LBD is expected to be similar to *learned* LBD; A valuable clause for a *learned* worker is also valuable for *imported* workers. However, the result reveals that more than 90% of *learned* clauses are re-evaluated worse at *imported* worker. This means a significant change in the LBD values after sharing. This observation can be explained with the previous *used%* analysis; The possibility of the clause being updated is low because the utilization of *imported* clauses is limited.

Through these experiments, we conclude that LBD is not enough for the clause quality evaluation in the parallel environment. We suggest that it is attributed to LBD's search state dependence. Under such conditions, we suggest using another clause evaluation metric that is independent of the search state.

5 Proposal method and implementation

Our proposed method, PaCS, evaluates clauses using a metric originally proposed for the clause deletion task [21]. The metric, namely average external edge weight (AEEW), measures the weight of the edges in the VIG of the input SAT problem. The sequential solver that used AEEW as its primary criterion for clause evaluation demonstrated competitive performance with LBD. The value of AEEW for a clause c is expressed as follows:

$$AEEW(c) := avg(\{w(e)/|c| : e = (v_i, v_j) \in E, v_i \in c, v_j \notin c\})$$

This concept denotes the strength (presented as the weight of edge) of the connection between the nodes v_i and v_j, where v_i belongs to the learnt clause c ($v_i \in c$) and v_j is the neighboring variable of v_i that does not belong to c ($v_j \notin c$). E denotes all edges in G, $\forall e \in E$, and $e = (v_i, v_j)$ indicates that edge e connects to variables v_i and v_j. Function $w(e)$ returns the weight of edge e in G, which is defined in Section 2.4. $|c|$ denotes the size of c, and the function $avg(X)$ calculates the arithmetic mean of set X, defined as $avg(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$, where n denotes the count of elements in X ($|X|$). A higher AEEW value (heavier edge on average or smaller size) indicates a higher quality.

We suggest that AEEW can measure a clause's quality from a new perspective: "the possibility of determining the Boolean values of variables in the evaluating clause". A higher AEEW value indicates either a heavier average edge weight or fewer variables in the clause. A heavy edge implies that the pair of variables belonging to the edge is contained in a short clause and/or many clauses as defined by the weight. When the Boolean value of one variable is determined, the Boolean value of the other variable is more likely to be determined through propagation. Also, the fewer variable clauses (shorter clauses) have a higher possibility of determining all variables in the clause. Thus, higher AEEW clauses have a better chance of determining their variables' Boolean values easily. Consequently, this allows AEEW to positively evaluate clauses that are more likely to be satisfied, initiate unit propagation, or induce conflicts.

We focused on AEEW's search-independent property, which is attributed to its definition using the graph structure of INPUT CNF. Thus, we propose PaCS using AEEW as a metric for selecting sharing clauses in parallel SAT solvers. This has the potential to address the challenges LBD faces in parallel environments. Algorithms 1 and 2 show how PaCS determines which clauses to share. For implementation, we used ParKissat-RS as the base solver. ParKissat-RS adopts a worker-sharer parallel architecture. Workers add acquired learnt clauses that meet the criteria to a buffer for sharing. In the default configuration, the criterion has an LBD value of two or less. The sharer broadcasts all clauses in the buffer to the workers at regular intervals. We modified both procedures in the worker and sharer and adopted a two-step selection in PaCS. Algorithm 1 describes the procedure in a worker, the first-step selection. A clause is stored in *buffer* and forwarded to sharer if its size is less than or equal to a predefined value *limit_size* or its LBD values less than or equal to two. The *limit_size* and LBD ≤ 2 serve as the initial screening criteria to decrease the computational cost of calculating and sorting the AEEW values in the sharer procedure. In the case of LBD and size, screening is not necessary by setting a constant threshold (e.g.

size ≤ 8) among instances. However, it is difficult to set a constant threshold by AEEW since its value varies largely from instance to instance. Therefore, we set initial screening to shortlist millions of obtained clauses by size to a manageable amount and then sort the remaining clauses according to the value of AEEW. Furthermore, we added the LBD ≤ 2 criterion for the following *cnt* implementation. Then, the procedure in sharer serves as the second screening, as described in Algorithm 2. One worker constructed the instance graph in a parallel environment on shared memory, and then, upon completion, other workers or sharers used it. The AEEW values are calculated for all clauses in the buffer and then sorted. The sorting is in descending order of AEEW, ascending order of LBD if the AEEW values are equivalent, and ascending order of size if the LBD values are equivalent. Higher AEEW clauses are shared more preferentially. The solver counts *cnt* as the number of clauses in the buffer with an LBD value less than or equal to two. This *cnt* constrains the number of clauses to be shared at line 6; this helps to exclude the effect of the total number of sharing clauses and observe only the effect of the change in selection criteria.

This implementation allows us to isolate and compare the effects of changing the criteria for selecting shared clauses. Subsequently, the sharer executes learnt clause sharing at predetermined intervals every 0.5 s. The selected clauses are broadcast to parallel workers from the top of the sorted clauses.

Algorithm 1 Clause sharing procedure of PaCS in worker.

```
Require: obtained learnt clause $c$, LBD criteria $limit\_size$
1: if $c.size \leq limit\_size$ or $c.lbd \leq 2$ then
2:     $buffer \leftarrow c$
3: end if
4: submit $c$ to $buffer$
```

Algorithm 2 Clause sharing procedure of PaCS in sharer.

```
Require: all buffered learnt clauses $C$, counter $cnt$, instance graph $G$
1: for clause $c$ in $C$ do
2:     calculate AEEW value for clause $c$ using $G$
3:     if $c.lbd \leq 2$ then $cnt{+}{+}$
4: end for
5: sort($buffer$)
6: broadcast the top $cnt$ clauses in the $buffer$ according to the sorted order.
```

6 Performance Evaluation

6.1 Experiment setup

We evaluated the performance of the solver using the PaCS method and compared its performance with that of the base solver, ParKissat-RS, using LBD. We compared four types of solvers in 16 parallel environments (16 workers): Base – the default ParKissat-RS. Size (size ≤ 1 or 8) – share clauses whose size is less than or equal to eight (same as presented in Section 4.1). Random – randomly select clauses from those whose sizes are less than 100, up to the quantity corresponding to the number of clauses with LBD values of one or two; PaCS (size $\leq x$) – select clauses using PaCS from those whose size is less than or equal to x (*limit_size*), up to the quantity corresponding to the number of clauses with LBD values of one or two.

The benchmark comprised 1200 instances from the main tracks of SAT competitions held between 2021 and 2023 (400 instances per year). We conducted the experiments on a computer with an AMD Threadripper Pro 3995WX processor (64 core) and 512 GB (128 GB 4 slots, DDR4-3200 MHz) RAM. We used the default ParKissat-RS implementation, adding the necessary functions for PaCS, and only changed the timeout option. No other options for the running solvers. We evaluated the solver's performance based on the number of instances solved within a time limit of 5000 s on the CPU clock and the PAR-2 score, which represents the mean time required to solve an instance with an additional penalty of 5000 s for each unsolved instance.

6.2 Evaluation result

Table 3 summarizes the results of our experiments, in which we compared base, random, size $\leq$ 1, 8, and PaCS, whose initial screening size is between 5 to 9. SC21, SC22, and SC23 present the benchmark instance set from the SAT competition 2021-2023, respectively. SAT and UNSAT indicate the number of instances identified as satisfiable and unsatisfiable, respectively. Therefore, a higher number indicates a better result. The cactus plot in Figure 4 demonstrates the same results; however, the PaCS results are limited to size $\leq 5, 7$ for readability.

Table 3 Performance evaluation results corresponding to each solver.

	SC21		SC22		SC23		Total	PAR-2
Criterion	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT		
Base	153	162	158	153	124	141	891	2934
Random	154	156	159	148	125	137	879	3170
Size (size $\leq$ 1)	155	160	160	151	126	147	899	2931
Size (size $\leq$ 8)	154	167	159	160	**129**	141	910	2754
PaCS (size $\leq$ 5)	155	**170**	162	164	128	148	927	2612
PaCS (size $\leq$ 6)	155	169	**163**	164	124	**150**	925	2634
PaCS (size $\leq$ 7)	**157**	169	160	**166**	127	**150**	**929**	**2595**
PaCS (size $\leq$ 8)	154	168	161	164	127	148	922	2659
PaCS (size $\leq$ 9)	156	168	**163**	164	124	**150**	925	2642

The size-based selection solver performed marginally better than the LBD, which is consistent with the results of Section 4.1. However, the random one worsened the performance. The solver's performance using PaCS is overall better than that of the base solver using LBD, particularly with the initial size seven screenings. It solved 38 more instances (+4.2%) and achieved an average PAR-2 score improvement of 339 (-11.6%). More improvement was observed in the UNSAT instances. At a high level, UNSAT instances require good learning for their proof, and we assume that PaCS contributed to the identification of these valuable clauses for all workers. This improvement can be attributed to two properties of PaCS. PaCS can share valuable clauses for all parallel workers irrespective of their search states because it depends only on the graph structure converted from the CNF of the input problem. This implies that PaCS can assess the general usefulness of clauses across parallel workers, whereas LBD indicates the value of a worker's search state. Furthermore, PaCS can favorably select clauses whose variables are more likely to be propagated because it highly values heavier edges in VIG. After determining the Boolean values of variables in the clauses, it can identify the clauses that cause conflict more frequently.

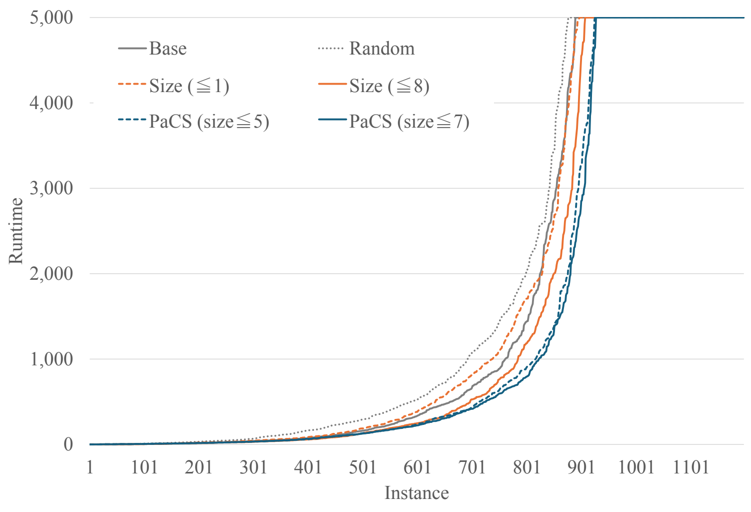


Figure 4 Performance evaluation results in cactus-plot. PaCS results are limited to size $\leq 5, 7$ for readability.

7 Conclusion

This study focuses on the clause-sharing strategy for parallel SAT solvers to improve them. First, we investigated the performance of LBD, the current popular metric, in parallel clause sharing. Preliminary experiments showed that LBD is not optimal for clause evaluation in a parallel environment. Therefore, we propose a novel clause-sharing method, Parallel Clause sharing based on graph Structure, PaCS. It can evaluate the common quality for all parallel workers using the graph structure derived from the input CNF of the SAT problems. The performance evaluation experiments demonstrated that PaCS outperforms the state-of-the-art parallel solver using LBD. These results showed the potential for enhancing the clause-sharing strategy of parallel solvers by leveraging the graph structure inherent in SAT problems. Furthermore, we believe that this study opens avenues to reinterpret and understand the quality of learnt clauses more deeply. The quality of clauses has often been evaluated by their size (including substantial size measured by LBD). We argue that AEEW is the extended concept of size, which encompasses what size implies. Short clauses have a higher possibility of determining Boolean values, which are more likely to induce unit propagation and conflict, contributing to the search. The AEEW can assess the possibility directly using the weight of edges in the VIG.

The following items are for future work. First, there is potential for improving performance by refining the implementation. The construction of graphs entails a certain duration, which may span up to several hundred seconds in one worker, depending on the problem. Further enhancements to the data structures can contribute to performance improvements. The current algorithm includes pre-defined parameters, such as initial screening before submission to a buffer, which can be optimized. The second entails exploring adaptive strategies, such as altering selection criteria or the number of clauses to be shared. Third, large-scale parallel experiments may be studied further. Conducting these experiments may validate the scalability and practical applicability of our proposal method. The fourth is to replace LBD with PaCS for all use in parallel solvers, for example, clause deletion strategy in each parallel worker and optimization of the number of sharing clauses without LBD. Fifth, we would like to explore the theoretical justification and investigation of the implications of AEEW values, the possibility of propagation. Finally, in-depth investigations of the relationship between clause quality and graph structure from a broad perspective can contribute to understanding the behavior of SAT solvers and the quality of learnt clauses. These future works aim to optimize and expand the application of PaCS and explore new frontiers of clause evaluation.

References

1 Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 410–423, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

2 Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 200–213, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

3 Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezing and reactivating learnt clauses. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 188–200, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

4 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, July 2009. Morgan Kaufmann Publishers Inc.

5 Gilles Audemard and Laurent Simon. Lazy clause exchange policy for parallel sat solvers. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 197–205, Cham, 2014. Springer International Publishing.

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

7 Tomáš Balyo, Peter Sanders, and Carsten Sinz. Hordesat: A massively parallel portfolio sat solver. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 156–172, Cham, 2015. Springer International Publishing.

8 Roberto J. Bayardo and Robert C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 203–208. AAAI Press, 1997.

9 Armin Biere and Andreas Frohlich. Evaluating cdcl restart schemes. In *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 1–17. EasyChair, 2019. doi:10.29007/89dw.

10 Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003. doi:10.1016/S0167-8191(03)00068-1.

11 Curtis Bright, Ilias Kotsireas, Albert Heinle, and Vijay Ganesh. Complex golay pairs up to length 28: A search via computer algebra and programmatic sat. *Journal of Symbolic Computation*, 102:153–172, 2021. doi:10.1016/j.jsc.2019.10.013.

12 Wenjing Chang, Guanfeng Wu, and Yang Xu. Adding a lbd-based rewarding mechanism in branching heuristic for sat solvers. In *2017 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, pages 1–6, 2017. doi:10.1109/ISKE.2017.8258780.

13 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. `doi:10.1145/368273.368557`.

14 Andrea Ferrara, Guoqiang Pan, and Moshe Y. Vardi. Treewidth in verification: Local vs. global. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 489–503, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

15 Carla P Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *15th national conference on artificial intelligence and 10th conference on Innovative applications of artificial intelligence*, pages 431–437. AAAI, 1998.

16 Youssef Hamadi, Said Jabbour, and Jabbour Sais. *Control-Based Clause Sharing in Parallel SAT Solving*, pages 245–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. `doi:10.1007/978-3-642-21434-9_10`.

17 Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010. `doi:10.3233/SAT190070`.

18 Marijn J H Heule, Oliver Kullmann, and Victor W Marek. Solving and verifying the boolean pythagorean triples problem via Cube-and-Conquer. In *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer. `doi:10.1007/978-3-319-40970-2_15`.

19 Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997. `doi:10.1126/science.275.5296.51`.

20 Yoichiro Iida, Tomohiro Sonobe, and Mary Inaba. Structural impact of learnt clauses in sat solvers: An empirical analysis. In *Pragmatics of SAT*, 2023.

21 Yoichiro Iida, Tomohiro Sonobe, and Inaba Mary. Wance: Learnt clause evaluation method for sat solver using graph structure. In *The 18th Learning and Intelligent Optimization Conference (LION 2024)*, 2024. submitting and under review.

22 Saïd Jabbour, Jerry Lonlac, Lakhdar Saïs, and Yakoub Salhi. Revisiting the learned clauses database reduction strategies. *International journal of artificial intelligence tools: architectures, languages, algorithms*, 27(08):1850033, 2018. `doi:10.1142/S0218213018500331`.

23 Sima Jamali and David Mitchell. Centrality-based improvements to cdcl heuristics. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 122–131, Cham, 2018. Springer International Publishing.

24 George Katsirelos and Laurent Simon. Eigenvector centrality in industrial sat instances. In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 348–356, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

25 Nadjib Lazaar, Youssef Hamadi, Said Jabbour, and Michèle Sebag. Cooperation control in parallel sat solving: a multi-armed bandit approach. Technical report, Institut National de Recherche en Informatique et en Automatique, 2012.

26 Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. Painless: A framework for parallel sat solving. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 233–250, Cham, 2017. Springer International Publishing.

27 Jia Hui Liang, Hari Govind V.K., Pascal Poupart, Krzysztof Czarnecki, and Vijay Ganesh. An empirical study of branching heuristics through the lens of global learning rate. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 119–135, Cham, 2017. Springer International Publishing.

28 J.P. Marques-Silva and K.A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. `doi:10.1109/12.769433`.

29 M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001. `doi:10.1145/378239.379017`.

30 Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 252–268, Cham, 2014. Springer International Publishing.

31 Zack Newsham, William Lindsay, Vijay Ganesh, Jia Hui Liang, Sebastian Fischmeister, and Krzysztof Czarnecki. Satgraf: Visualizing the evolution of sat formula structure in solvers. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 62–70, Cham, 2015. Springer International Publishing.

32 Nicolas Prevot, Mate Soos, and Kuldeep S. Meel. Leveraging gpus for effective clause sharing in parallel sat solving. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 471–487, Cham, 2021. Springer International Publishing.

33 Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 518–534, Cham, 2021. Springer International Publishing.

34 Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26:521–532, September 1996.

35 Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. Community and lbd-based clause sharing policy for parallel sat solving. In *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 11–27, Cham, 2020. Springer International Publishing.

36 Richard Williams, Carla Gomes, and Bart Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *Theory and Applications of Satisfiability Testing 2003*, 2003.

37 Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger, Jia Hui Liang, Krzysztof Czarnecki, and Vijay Ganesh. The effect of structural measures and merges on sat solver performance. In *Principles and Practice of Constraint Programming*, pages 436–452, Cham, 2018. Springer International Publishing.

Global Benchmark Database

Markus Iser
Karlsruhe Institute of Technology, Germany

Christoph Jabs
HIIT, University of Helsinki, Finland

Abstract

This paper presents Global Benchmark Database (GBD), a comprehensive suite of tools for provisioning and sustainably maintaining benchmark instances and their metadata. The availability of benchmark metadata is essential for many tasks in empirical research, e.g., for the data-driven compilation of benchmarks, the domain-specific analysis of runtime experiments, or the instance-specific selection of solvers. In this paper, we introduce the data model of GBD as well as its interfaces and provide examples of how to interact with them. We also demonstrate the integration of custom data sources and explain how to extend GBD with additional problem domains, instance formats and feature extractors.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Design and analysis of algorithms; Information systems → Information integration

Keywords and phrases Maintenance and Distribution of Benchmark Instances and their Features

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.18

Supplementary Material *Software (GBD Source Code)*: `https://github.com/Udopia/gbd` [17]
archived at `swh:1:dir:37fb54c7420fd71b6c4930bcf7e78d5129325a6d`
Software: `https://github.com/Udopia/gbdc` [16]
archived at `swh:1:dir:705258b1bd5c3415069fbd156759945a8ad725c0`
Software: `https://github.com/Udopia/gbdeval` [15]
archived at `swh:1:dir:159fae7ac54d9f678ee38ed882dae693cd5f14cf`
Dataset: `https://github.com/Udopia/gbd-data` [14]
archived at `swh:1:dir:1764eef19455a0bfdb6389a04a055977338304ff`

Funding *Christoph Jabs*: Work financially supported by Research Council of Finland grant 356046.

Acknowledgements Many thanks to all the people who have supported GBD, either by contributing code or data, or by providing infrastructure. Special thanks also go to the early adopters whose successes and feedback have been and continue to be inspiring and motivating.

1 Introduction

The idea to create Global Benchmark Database (GBD) arose from the need to make available and sustainably maintain benchmark instances and their metadata for the problem of propositional satisfiability (SAT). To this end, we specified a simple hash function for *instance identification* to serve as the primary key for SAT instance data sets. A proof of concept was presented and discussed at the 2018 Pragmatics of SAT (POS) workshop, demonstrating the specification of a SAT instance identifier and methods for labeling and querying for instances [19]. From there, GBD has matured into a comprehensive suite of tools for provisioning and sustainably maintaining benchmark instances of various problem domains and their metadata. Supported problem domains include propositional satisfiability (SAT), maximum satisfiability (MaxSAT), and pseudo-Boolean optimization (PBO).

In essence, the purpose of GBD is to act as a conduit between data science and empirical research on NP-hard problem classes by facilitating the seamless integration of benchmark data into existing workflows. GBD provides tools for distributing benchmark instances and feature databases, as well as tools for transforming instances and extracting instance features. It also provides a set of prebuilt feature databases for identifying instance equivalence classes, categories, or labels, as well as analytical instance features. Examples of successful applications of GBD include the sanitization and selection of benchmarks for SAT competitions [13], domain-specific solver evaluations [8], and the analysis of solver portfolios and solver prediction models [2]. Moreover, the authors of the latest award-winning state-of-the-art SAT solvers use GBD in their empirical evaluations. [11, 25]. GBD is open source and available under the MIT license [18].

In this paper, we describe the conceptual design of the GBD data model and present the interfaces for interacting with GBD, as well as the extension capabilities provided by the data model, including concrete examples of their use. We start with a brief description of the GBD data model and query language in Section 2. Section 3 presents the GBD interfaces and gives concrete examples of how to use GBD in practice. Finally, Section 4 explains how to extend GBD with additional problem domains, instance formats, and feature extractors. We conclude with a summary and an outlook on future work in Section 5.

2 Data Model and Queries

Conceptually, GBD consists of an extensible set of contexts $\mathcal{C} = \{C_0, C_1, \dots\}$. Each context represents a problem domain, or more specifically, a particular instance format of a problem domain. A context is defined as a tuple $C = (\mathcal{X}, \mathsf{id})$, where $\mathcal{X}$ is the set of benchmark instances and $\mathsf{id} : \mathcal{X} \to \mathsf{PK}$ is a context-specific instance identification function that maps each benchmark instance to an instance identifier unique within the context. Contexts are instantiated with data sources $D = \{\mathcal{F}_1, \mathcal{F}_2, \dots\}$ which provide instance features $\mathcal{F}_i \subseteq \mathsf{PK} \times 2^{\Sigma^*}$ assigning values to benchmark instances, where values are represented by strings over an alphabet Σ.[1]

Feature Types

In practice, GBD distinguishes between one-to-one and one-to-many features. One-to-one features are defined as those that provide a single value per instance, with a default value associated with them. In contrast, one-to-many features can have zero to any number of values per instance. One-to-one features facilitate the initial setup and subsequent maintenance, as they are initialized to the default value automatically and subsequent calls to gbd set overwrite any previous settings. In contrast, the values of one-to-many features accumulate over time and must be explicitly cleared. Example 1 illustrates the practical use of the different feature types in the context of cnf instances.

▶ **Example 1.** Let cnf $:= (\mathcal{X}, \mathsf{id})$ be the context where $\mathcal{X}$ is the set of SAT benchmark instances in DIMACS CNF format, and let a data source meta.db $:= \{\mathsf{track}, \mathsf{result}, \dots\}$ be given, which provides the cnf instance features track and result. In the example, the feature track indicates the competition tracks in which the instance was used and is modeled as a one-to-many feature, while result indicates the satisfiability of the instance and is defined as a one-to-one feature with the default value “unknown”.

[1] The data sources used in the examples of this section are available at `https://benchmark-database.de`.

Context Mappings

For any two contexts $C_i = (\mathcal{X}_i, \mathsf{id}_i)$, $C_j = (\mathcal{X}_j, \mathsf{id}_j)$ with instance identifiers $\mathsf{PK}_i, \mathsf{PK}_j$, a data source can provide a special feature $\mathcal{F} \subseteq \mathsf{PK}_i \times \mathsf{PK}_j$ that establishes a relationship between instances in $\mathcal{X}_i$ and $\mathcal{X}_j$. In GBD, such features are referred to as context mappings. Context mappings can be used to map instances from different problem domains to each other if there is a reduction procedure that transforms instances from one domain to instances in the other domain. Example 2 illustrates the practical use of context mappings to relate instances of the propositional satisfiability problem to instances of the k-independent set problem.

▶ **Example 2.** Given the context of cnf instances of the SAT problem and the context kis of graph-instances of the k-Independent Set problem. Any SAT instance can be transformed to a graph and a number k in such a way that the satisfiability of the SAT instance is equivalent to the existence of an independent set of size k in the graph [23]. A context mapping feature can be used to map instances in cnf to their corresponding instances in kis.

Context mappings are useful not only for relating instances from different problem domains, but also for relating instances from different contexts within the same problem domain. Application scenarios include relating different instance formats, analyzing different encodings of application instances, or distinguishing sanitized or otherwise preprocessed instances from their original counterparts (cf. Use case 9 below).

Queries

GBD provides a query language designed to filter instances of a given context by specifying constraints over the instance features. A GBD query is a simple constraint c, or a compound constraint c_1 `and` c_2, or c_1 `or` c_2 for sub-constraints c_1, c_2, and parentheses can be used to indicate precedence. A simple constraint is of the form $f \circ e$ with a feature name f, a value e, and an operator $\circ \in \{$`=, !=, <, >, <=, >=, like, unlike`$\}$ with the usual semantics. For SQL-inspired operators $\circ \in \{$`like, unlike`$\}$, a constraint can also be of the form $f \circ \%e$, $f \circ e\%$, or $f \circ \%e\%$ to indicate postfix, prefix, or infix constraints, respectively. Value e can also be a term, where a term is either a constant number, or a feature name in parentheses (f')[2], or a compound term $(t_1 \circ t_2)$ from arithmetic operators $\circ \in \{$`+,-,*,/`$\}$ and terms t_1, t_2.

▶ **Example 3.** Given the cnf context and the data source meta.db providing the instance feature track, a query can be of the form track = `main_2023` to filter for instances from the Main track of the 2023 SAT competition.

▶ **Example 4.** Given the cnf context and the data source meta.db providing the instance features track and filename, a query can be of the form track = `anni_2022` and filename like `rphp%` to filter for instances from the Anniversary track of the 2022 SAT competition whose filenames begin with the string `rphp`.

▶ **Example 5.** Given the cnf context and the data source base.db providing the instance features variables and clauses, a query can be of the form variables > `(clauses)` to filter for instances with more variables than clauses.

[2] Parentheses are used here to disambiguate terms and character strings.

3 Applications

This section introduces the gbd-tools package, which is available in the Python Package Index (PyPI) and can be installed using the command pip install gbd-tools. The package provides the command line tool gbd, a web service, and the Python interface class GBD. We start with instructions on how to configure gbd-tools in Section 3.1, and then present use cases of the command line tool gbd in Section 3.2. The web service and Python interface are presented in Sections 3.3 and 3.4, respectively.

3.1 Data Source Configuration

Data sources for instantiating GBD are specified as a list of files, where each file can be either a sqlite3 database created by GBD, or a csv file for importing data from other sources. The requirements for the csv files are that they have a header line containing the feature names, and they must provide the hash column containing the instance identifiers. Example 6 shows a typical example of integrating two data sources before analyzing runtime experiments.

▶ **Example 6.** Let GBD be instantiated with the two data sources meta.db and runtimes.csv. The file meta.db provides the feature family, indicating the domain of each instance. The file runtimes.csv provides the solver runtimes baseline and incumbent. The instance identifier in the hash column facilitates the integration of the two data sources and thus the domain-wise analysis of experimental results.

Data sources can be registered by setting the environment variable GBD_DB to a colon-separated list of paths. It is also possible to specify or override the data sources with the option -d/--db on each call of the gbd command-line tool. If a data source does not exist, GBD offers to create it. Ready-made feature databases are available for download from `https://benchmark-database.de`. The gbd info command can be used to display the registered data sources, their names, and the features provided. The name of a data source is automatically generated from its filename by removing the extension, and is needed in some parameters and queries to explicitly refer to a specific data source (cf. Use cases 3 and 7).

Each data source is associated with a context, so a data source can only provide features for exactly one context. Each context is identified by a name, e.g., opb for the context of pseudo-Boolean optimization instances. The set of available contexts, including their names and descriptions, can be displayed with the command gbd info -c. To bind a data source to a context, its filename is prefixed with the context name to which the data belongs (cf. Use cases 9 and 10). Otherwise, the data source is treated as belonging to the default context cnf.

When features from different contexts are queried simultaneously, a context mapping feature is used to create a relational join between the instance identifiers of the different contexts. The naming convention for context mapping features is to_{cxt}, where cxt is a context name. The to_{cxt} feature maps the identifiers of the context of the data source in which it resides to the identifiers of the context named by cxt. GBD recognizes context mapping features according to this naming convention and generates the appropriate foreign key relationships to automatically join features from different contexts.

3.2 Command-Line Tool gbd

In this section, we present the command line tool gbd and its subcommands for initializing GBD with benchmark instances, querying for instances and features, extracting features from instances, and transforming instances. The full set of subcommands can be viewed with gbd --help. In the use cases presented in this section, we assume that the cnf data sources meta.db[3] and mylocal.db (to be initialized in the following) are configured as data sources by the environment variable GBD_DB, and that /path/to/instances contains the benchmark instances from SAT Competition 2023.[4]

3.2.1 Database Initialization

In order to use GBD to organize your own experiments, it is necessary for GBD to know where the locally available benchmark instances are located. Integrating a set of paths to benchmark instances into GBD requires computing the instance identifiers and storing the instance paths in a GBD data source. In this process, GBD creates the reserved features local and filename to associate the local paths and filenames with the respective identifiers. Note that projecting to an instance identifier automatically eliminates duplicates. The initialization process may require a considerable amount of time, even for a moderate number of instances, as is the case with SAT competition benchmark sets. However, it is only required once for the registration of benchmarks.

Use case 1 illustrates initializing GBD with a number of benchmark instances present on the local volume using the gbd init local command. The -j/--jobs option is used to set the number of parallel jobs, the --target option is used to specify the name of the database in which to create the features, and the parameter after the local subcommand is used to specify the path to the benchmark instances. The gbd init local command recursively searches the specified directory for benchmark instances using the file extensions associated with the context of the target database.

Use case 1 Local database initialization.

```
gbd init -j16 --target mylocal local /path/to/instances
```

3.2.2 Queries for Instances and Features

The gbd get command is used to filter for specific instances and retrieve their features. A comprehensive set of options can be displayed with gbd get --help. Filtering is typically done with GBD queries, and a list of features to be returned is given by the -r/--resolve option. Use case 2 illustrates how to filter for cryptographic instances in the SAT Competition 2023 benchmark set and return their local paths.

Use case 2 Filtering and feature resolution.

```
gbd get "track=main_2023 and family like crypto%" -r local
```

Feature names from different databases may overlap in some configurations. In this case, GBD picks the feature values from the first database in the list that provides the feature. To override this behavior, feature names can be prefixed with the database name, separated by a colon. Use case 3 shows how to filter by the feature track from the database meta and select the feature local from the database mylocal.

Use case 3 Explicit feature database.

```
gbd get "meta:track=main_2023" -r mylocal:local
```

[3] https://benchmark-database.de/getdatabase/meta
[4] https://benchmark-database.de/?track=main_2023

To control the handling of multiple values per hash, the -c/--collapse option can be used, as shown in Use case 4. The command collapses the values of the feature local to a single value by using the minimum value.

■ **Use case 4** Collapse.

```
gbd get "track=main_2023" -r mylocal:local -c min
```

For grouping instances by a feature other than the instance identifier, the option -g/--group can be used as illustrated in Use case 5. The command groups the instances by the feature isohash and selects a local path for each group using the minimum function.

■ **Use case 5** Grouping.

```
gbd get "track=main_2023" -r local -c min -g isohash
```

3.2.3 Manual Data Acquisition

In addition to subcommands for creating, deleting, renaming, and copying features, the gbd command line tool provides a subcommand for manually setting feature values. Use case 6 demonstrates the use of the gbd set command to set the value of the family feature to hardware-verification for all instances whose filename starts with the string manol.

■ **Use case 6** Setting feature values.

```
gbd set family=hardware-verification "filename like manol%"
```

3.2.4 Instance Feature Extraction

Feature extractors are accessible through subcommands of gbd init, which can be listed with gbd init --help. Each feature extractor is bound to a set of contexts and can only be executed on instances from those contexts. Multiple feature extractors are provided by the gbdc extension module which are documented at `https://udopia.github.io/gbdc`. Note that ready-made feature databases are available for download from our instance of the GBD web interface (cf. Section 3.3).

Use case 7 illustrates the use of the subcommand gbd init base to extract default features denoted as base from a set of cnf benchmark instances. In the example, the option -j/--jobs is used to specify the number of parallel jobs to use for the feature extraction. The option --target is used to specify the name of the database in which to create the features. This requires that a database with the name mybase is registered as a data source. The parameter after the subcommand base is used to specify the query for the instances to extract features from.

■ **Use case 7** Base feature extraction.

```
gbd init -j16 --target mybase base "track=main_2023"
```

It is often necessary to determine which instances are isomorphic to each other. The isohash feature is a hash value that over-approximates the class of isomorphic instances and can be used to identify and group isomorphic instances and to subsequently eliminate them from benchmark sets (cf. Use case 5). Use case 8 illustrates how to use the gbd init isohash command to compute isohash for a set of cnf benchmark instances.

■ **Use case 8** Isohash calculation.

```
gbd init -j16 --target meta isohash "track=main_2023"
```

3.2.5 Instance Transformation

GBD also provides instance transformers accessible via subcommands of gbd transform. A complete list of available instance transformers can be displayed with gbd transform --help. Instance transformers are functions that transform an instance of one context into another instance of a different context. Additionally, an instance transformer automatically creates a context mapping that relates the instances in both contexts, and can store additional features in the database. Instance transformers are implemented as a special type of feature extractor that, in addition to creating instance features, also creates a new benchmark instance in the target context. Several instance transformers are provided by the gbdc extension module and are documented in the gbdc documentation.

Use case 9 illustrates the use of the sanitize transformer for cnf instances. The command transforms the instances in the source context cnf to instances in the target context sancnf and stores their mapping and local path in the database sancnf_local.db.

Use case 9 Instance sanitizer.

```
gbd transform --source cnf --target sancnf_local sanitize
```

Note that sancnf is an example of a context that exists only to distinguish between sanitized and unsanitized instances and to use the context mapping feature to check if identifiers have changed. In most practical use cases, the sanitized instances will be used to replace the unsanitized instances in the original cnf context.

An example of a real transformation between different problem domains is the transformation of cnf instances into k-Independent Set (kis) instances, as described in Example 2. Use case 10 illustrates the use of the cnf2kis transformer to transform the instances in the source context cnf into instances in the target context kis. The mapping of the instances in both contexts is stored in the database kis_local.db.

Use case 10 Transforming CNF to KIS.

```
gbd transform --source cnf --target kis_local cnf2kis
```

3.3 Web Interface

The GBD web interface is implemented as a RESTful web service [7] that provides access to the benchmarks and databases of the environment it runs in, and can be started with the command gbd serve. While the GBD command-line tool operates independently of the web service, the web service uses the same GBD API and database configuration as the command-line tool. Our instance of the GBD web interface, accessible via `https://benchmark-database.de`, is configured to run behind an Nginx reverse proxy [24] and is hosted in a Docker container [21]. At the time of writing, we provide access to more than 100 000 benchmark instances from the cnf, wcnf, and opb contexts, with prebuilt feature databases for them.

3.4 Python Interface

The GBD Python API is wrapped in the class gbd_core.api.GBD and its documentation can be found at `https://udopia.github.io/gbd`. With the Python API, GBD data sources can be directly integrated into Python scripts and Jupyter notebooks to be used in evaluation and analysis. Use case 11 illustrates the use of the Python API to query for instances and features. Line 2 creates a GBD object that connects to the GBD data sources base and meta, which are passed to the constructor as a list of paths. Line 3 creates a list of feature names from the base data source and adds the feature family from the meta data source. Line 4 queries for the given features of cnf instances from the Main track of SAT Competition 2023. The result is returned as a Pandas [20] DataFrame for further analysis.[5]

Use case 11 GBD Python interface.

```
1 from gbd_core.api import GBD
2 with GBD(['gbd/meta.db', 'gbd/base.db']) as gbd:
3     feat = gbd.get_features('base') + ['family']
4     df = gbd.query("track=main_2023", resolve=feat)
```

4 Writing Extensions

The architectural design of GBD allows the integration of new contexts, feature extractors, and instance transformers. The extensibility of the system is achieved through the use of dictionaries, which serve as registries for contexts, feature extractors, and instance transformers. The following section describes the elements required to create such a registry entry. It is important to note that at the time of writing, these registries are hard-coded, which is a limitation in that it requires modification of these dictionaries within the source code of GBD. This issue will be addressed in future versions of GBD, where the registry will be moved to configuration files.

GBD contexts are managed in a dictionary data structure that is initialized in the gbd_core.contexts module. The dictionary is indexed with the respective *context name* and contains a pointer to the instance *identification function* and a list of valid instance *filename extensions*. The instance identification function is the function that assigns a unique instance identifier to each benchmark instance, which serves as the primary key in the context-specific feature databases. The list of valid instance filename extensions is used to locate benchmark instances in the local file system during the database initialization process. Once a context is created in the dictionary, context-specific databases can be populated using the context name.

A new feature extractor can be integrated into GBD by registering a feature extractor function in a dictionary data structure that is initialized in the gbd_init.feature_extractors module. The dictionary is indexed by the *name* of the feature extractor and contains a *list of the provided features*, which are tuples of feature names and default values. It also contains a pointer to the feature extractor *function* and a list of *contexts* in which it can be used. Registered feature extractors are automatically accessible by their names as subcommands of gbd init.

Instance transformers are managed in a dictionary data structure that is initialized in the gbd_init.instance_transformers module. An instance transformer function takes an instance in a *source context* and transforms it to a new instance in the *target context*. The dictionary is indexed by the *name* of the instance transformer and, similarly to feature extractors, contains a *list of the provided features*. Additionally, the dictionary contains a pointer to the instance transformer *function itself*, as well as a pointer to a function that generates the *name of the benchmark instance* to be created. Registered instance transformers are automatically accessible by their names as subcommands of gbd transform.

[5] We provide illustrative examples of benchmark data analyses, including portfolio analysis, category prediction, and category-specific ranking, accessible via `https://udopia.github.io/gbdeval/`.

5 Related and Future Work

SatLib was the first public collection of benchmark instances in the SAT problem domain [12], and similar collections have been created for other problem domains such as MaxSAT [1], Quantified Boolean Formulas (QBF) [9], SAT Modulo Theories (SMT) [4], and Mixed Integer Programming (MIP) [10]. SatEx is the first web-based framework for reproducible execution and evaluation of SAT solver experiments [26], followed by the EDACC framework [3]. Instance features were used to predict the fastest solver for an instance [27] and to reduce redundancy in experiments [22]. Aslib is a library of data sets for training and evaluating solver prediction models [5]. Meta-features such as the instance domain have been used to study specialized heuristic configurations [6].

GBD's data model, which is centered around instance identification functions, enables the data-driven study of NP-hard problem domains in an unprecedentedly sustainable manner. Future work includes the integration of more problem domains, more feature extractors, and more instance transformers. We also plan to improve the configuration capabilities of the data sources, including the ability to easily switch between different configurations, and to improve the extensibility of GBD by making it a matter of changing a configuration file whenever new problem domains, instance formats, feature extractors, and instance transformers are added. We also plan to improve the web interface to make it easier to use and more informative, and to provide a benchmark submission system. Finally, we plan to automate the feature extraction process to instantly provide feature databases for new benchmark instances, and add support for instance generators.

The purpose of coupling and providing instance data is to enable more complex data-driven analyses of the instance spaces of hard algorithmic problems, thus accelerating the growing use of explainable artificial intelligence methods. Future work in this area includes the application of machine-assisted hypothesis generation and testing, and the development of new methods for exploring, analyzing, and explaining algorithmic datasets.

References

1 Fahiem Bacchus. MaxSAT Formulas Library, 2006. URL: http://www.cs.toronto.edu/maxsat-lib/maxsat-instances/.

2 Jakob Bach, Markus Iser, and Klemens Böhm. A Comprehensive Study of k-Portfolios of Recent SAT Solvers. In *Intl. Conf. on Theo. and Appl. of Satisf. Test., SAT*, pages 1–18, 2022. doi:10.4230/LIPIcs.SAT.2022.2.

3 Adrian Balint, Daniel Gall, Gregor Kapler, and Robert Retz. Experiment Design and Administration for Computer Clusters for SAT Solvers, EDACC. *J. Satisf. Bool. Model. Comput.*, pages 77–82, 2010. doi:10.3233/SAT190078.

4 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library, SMT-LIB, 2016. URL: www.smt-lib.org.

5 Bernd Bischl, Pascal Kerschke, Lars Kotthoff, et al. ASlib: A Benchmark Library for Algorithm Selection. *Artif. Intel.*, pages 41–58, 2016. doi:10.1016/j.artint.2016.04.003.

6 Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström, and Laurent Simon. Seeking Practical CDCL Insights from Theoretical SAT Benchmarks. In *Intl. Joint Conf. on Artif. Intel., IJCAI*, pages 1300–1308, 2018. doi:10.24963/ijcai.2018/181.

7 Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

8 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artif. Intel.*, 2021. doi:10.1016/j.artint.2021.103572.

9 E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas Library, QBFLIB, 2005. URL: www.qbflib.org.

10 Ambros Gleixner, Gregor Hendel, Gerald Gamrath, et al. MIPLIB 2017: Data-driven Compilation of the 6th Mixed-Integer Programming Library. *Math. Program. Comput.*, pages 443–490, 2021. doi:10.1007/s12532-020-00194-3.

11 Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. Effective Auxiliary Variables via Structured Reencoding. In *Intl. Conf. on Theo. and Appl. of Satisf. Test., SAT*, pages 1–19, 2023. doi:10.4230/LIPICS.SAT.2023.11.

12 Holger Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *SAT 2000*, pages 283–292, 2000. URL: https://api.semanticscholar.org/CorpusID:17486963.

13 Markus Iser. Benchmark Compilation for SAT Competition 2023. In *Proc. of SAT Comp. 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023. URL: http://hdl.handle.net/10138/563824.

14 Markus Iser and Christoph Jabs. GBD Data Repository. Dataset, version 1.0., swhId: swh:1:dir:1764eef19455a0bfdb6389a04a055977338304ff (visited on 2024-08-02). URL: https://github.com/Udopia/gbd-data.

15 Markus Iser and Christoph Jabs. GBD Evaluation Scripts. Software, version 1.0., swhId: swh:1:dir:159fae7ac54d9f678ee38ed882dae693cd5f14cf (visited on 2024-08-02). URL: https://github.com/Udopia/gbdeval.

16 Markus Iser and Christoph Jabs. Global Benchmark Database (Extension Module). Software, version 1.0., swhId: swh:1:dir:705258b1bd5c3415069fbd156759945a8ad725c0 (visited on 2024-08-02). URL: https://github.com/Udopia/gbdc.

17 Markus Iser and Christoph Jabs. Udopia/gbd. Software, version 4.8.5., swhId: swh:1:dir:37fb54c7420fd71b6c4930bcf7e78d5129325a6d (visited on 2024-08-02). URL: https://github.com/Udopia/gbd.

18 Markus Iser and Christoph Jabs. GBD Tools, 2024. doi:10.5281/zenodo.11093597.

19 Markus Iser and Carsten Sinz. A Problem Metadata Library for Research in SAT. In *Proceedings of Pragmatics of SAT, POS*, pages 144–152, 2018. doi:10.29007/gdbb.

20 Wes McKinney et al. Data Structures for Statistical Computing in Python. In *Python in Science*, pages 51–56, 2010. doi:10.25080/majora-92bf1922-00a.

21 Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014.

22 Sibylle Möhle and Norbert Manthey. Better Evaluations by Analyzing Benchmark Structure. In *Pragmatics of SAT workshop, POS*, 2016. URL: http://www.pragmaticsofsat.org/2016/reg/POS-16_paper_4.pdf.

23 Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

24 Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008.

25 Dominik Pascal Schreiber. *Scalable SAT Solving and its Application.* PhD thesis, Karlsruhe Institute of Technology, KIT, 2023. doi:10.5445/IR/1000165224.

26 Laurent Simon and Philippe Chatalic. SatEx: A Web-based Framework for SAT Experimentation. *Electron. Notes Discrete Math.*, pages 129–149, 2001. doi:10.1016/S1571-0653(04)00318-X.

27 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intel. Res., JAIR*, pages 565–606, 2008. doi:10.5555/1622673.1622687.

On Limits of Symbolic Approach to SAT Solving

Dmitry Itsykson
Ben-Gurion University of the Negev, Beer-Sheva, Israel
On leave from Steklov Institute of Mathematics at St. Petersburg, Russia

Sergei Ovcharov
St. Petersburg State University, Russia

Abstract

We study the symbolic approach to the propositional satisfiability problem proposed by Aguirre and Vardi in 2001 based on OBDDs and symbolic quantifier elimination. We study the theoretical limitations of the most general version of this approach where it is allowed to dynamically change variable order in OBDD. We refer to algorithms based on this approach as $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms.

We prove the first exponential lower bound of $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms on unsatisfiable formulas, and give an example of formulas having short tree-like resolution proofs that are exponentially hard for $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms. We also present the first exponential lower bound for natural formulas with clear combinatorial meaning: every $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithm runs exponentially long on the binary pigeonhole principle BPHP_n^{n+1}.

2012 ACM Subject Classification Theory of computation → Proof complexity

Keywords and phrases Symbolic quantifier elimination, OBDD, lower bounds, tree-like resolution, proof complexity, error-correcting codes, binary pigeonhole principle

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.19

Funding *Dmitry Itsykson*: Supported by European Research Council Grant No. 949707.

1 Introduction

The Boolean satisfiability problem (SAT) is the decision problem, where given a CNF formula we are to decide whether it is satisfiable or not. The symbolic approach to SAT is to gradually transform the input formula into a special representation model in which the satisfiability problem can be easily solved. In our case, the representation model is an ordered binary decision diagram (OBDD) due to Bryant [4]. An ordered binary decision diagram (OBDD) represents a Boolean function as a branching program with two sinks such that on every path from the source to a sink, variables appear in the same order. This restriction on the order of variables allows handling of the diagrams very efficiently.

Assume that the input formula is $\varphi = \bigwedge_{i=1}^m C_i$. The naive symbolic approach is to choose some permutation σ of the set $[m]$ and iteratively compute OBDD representing $\bigwedge_{i=1}^k C_{\sigma(i)}$ for $k = 1, 2, \ldots, m$. Algorithms implementing this approach vary in the way of choosing a permutation σ (for example, it can be chosen dynamically and not straightaway) and in the way of choosing the order of variables in OBDD. Algorithms that choose the one order of variables in OBDD and do not change it during the execution we denote by $\mathrm{OBDD}(\land)$ algorithms. Algorithms that dynamically change the order in OBDD we denote by $\mathrm{OBDD}(\land, \mathrm{reordering})$ algorithms.

Aguirre and Vardi suggested a smarter symbolic approach called symbolic quantifier elimination [1, 19]. The key idea is that we have to obtain OBDD representation not of the formula $\varphi(x_1, x_2, \ldots, x_n)$ itself but rather $\exists x_1 \exists x_2 \ldots \exists x_n \varphi$. On one hand, the exterior quantifiers are not necessary to be applied since the satisfiability testing is easy for OBDD. But in some cases, one can move the quantifier in the middle of the formula. Namely, if all occurrences of a variable x_j are among clauses $C_{\sigma(i)}$ for $i \in [k]$, then the quantifier for x_j

can be put in front of $\bigwedge_{i=1}^{k} C_{\sigma(i)}$ and we can apply projection over x_j to the current OBDD. In other words, now the current OBDD represents not a conjunction of clauses $\bigwedge_{i=1}^{k} C_{\sigma(i)}$ but rather $\exists x_{i_1} \dots \exists x_{i_\ell} \bigwedge_{i=1}^{k} C_{\sigma(i)}$, where the variables $x_{i_1}, \dots, x_{i_\ell}$ do not appear in $C_{\sigma(i)}$ for $i \in \{k+1, \dots, n\}$. Algorithms implementing this approach vary on how they can choose the permutation σ (possibly in a dynamic way), variables orders in OBDDs, and also on the way of moving existential quantifiers inside the formula if it is permitted. Similarly, we call such algorithms by OBDD$(\wedge, \exists)$ and OBDD$(\wedge, \exists, \text{reordering})$ algorithms depending on whether it is allowed to change variable orders in OBDDs.

In this paper, we study theoretical lower bounds on the running time of OBDD$(\wedge, \exists, \text{reordering})$ algorithms. The running time of every such algorithm can be bounded below by the maximal size of constructed OBDDs. Our goal is to construct hard formulas for which any algorithm must construct an OBDD of exponential size not dependently on choosing a permutation σ, variable orders in OBDDs, and moving of existential quantifiers.

Previous results. The study of OBDD algorithms is highly connected with the study of OBDD-based proof systems initiated by Atserias, Kolaitis, and Vardi [3] and then continued by other researchers [14, 6, 5]. Lower bounds for OBDD algorithms follow from the derivation size lower bounds in corresponding proof systems (if such lower bounds are known). In contrast with DPLL and CDCL algorithms [2, 17] proving lower bounds on satisfiable instances for OBDD algorithms is not harder than proving lower bound of unsatisfiable instances. Indeed, if we know that φ is a hard unsatisfiable formula for OBDD$(\wedge, \dots)$ algorithms then the formula $x \vee \varphi$ that is obtained from φ by addition of the new variable x to all clauses of φ is a hard satisfiable formula for the same class of OBDD$(\wedge, \dots)$ algorithms. However, we do not know such a reduction in the other direction.

Lower bounds for OBDD$(\wedge, \text{reordering})$ algorithms on satisfiable formulas are rather easy to prove. Indeed, since at the end, the algorithm necessarily represents the initial formula, it is enough to construct a family of Boolean functions that have small CNF representations but require OBDD of exponential size for every order of variables. For example, satisfiable Tseitin formulas have such property [10]. Proving a lower bound for OBDD$(\wedge, \text{reordering})$ algorithms on unsatisfiable formulas is harder since at the end of the execution the OBDD represents the constant false and hence it has a constant size. However, exponential lower bounds for OBDD$(\wedge, \text{reordering})$ algorithms on unsatisfiable formulas are implied from the lower bound for corresponding OBDD based proof system; for example, unsatisfiable Tseitin formulas on expanders and the pigeonhole principle are hard for OBDD$(\wedge, \text{reordering})$ algorithms [14].

At the same time unsatisfiable and satisfiable Tseitin formulas [14] and the pigeonhole principle [7, 18] are easy for OBDD$(\wedge, \exists)$ algorithms.

Lower bounds for running time of OBDD$(\wedge, \exists)$ algorithms on unsatisfiable formulas follow from the lower bounds of the derivation complexity in dag-like and tree-like OBDD based proof systems [15, 20]. Unfortunately, the mentioned lower bounds' proofs rely significantly on the fact that all OBDDs use the same variable order. One more disadvantage of these results is that the hard formulas are very artificial and they were constructed especially for the proof of lower bounds.

Buss, Itsykson, Knop, and Sokolov showed that the ability to change order in OBDDs makes OBDD proof systems stronger [6]. One can use the same technique to show that the same is true for OBDD algorithms.

The paper [14] gives an exponential lower bound on the running time of OBDD$(\wedge, \exists, \text{reordering})$ algorithms on satisfiable formulas encoding that x is a codeword of the specific linear code. The question about lower bounds for OBDD$(\wedge, \exists, \text{reordering})$ algorithms on unsatisfiable formulas was open before this paper.

Some restricted lower bounds follow from the result of the paper [5] for corresponding restricted proof systems. Namely, unsatisfiable Tseitin formulas are hard for $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms if we bound the number of quantifiers that can be moved inside the formula; recall that such formulas are easy without this restriction. Also, the paper [5] gives an example of unsatisfiable formulas that are hard for $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms if it is allowed to use only a small number (at most $c \log n$, where c is a small constant and n is the number of variables) of orders.

Our results. In this paper, we give two families of unsatisfiable formulas and we prove for them that they require exponential running time of $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms.

The first family is based on a combination of hard satisfiable formulas for $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms and hard unsatisfiable formulas for $\mathrm{OBDD}(\land, \mathrm{reordering})$ algorithms. The proof strategy is the following. The algorithm either does not apply projections and, thus, it simulates an $\mathrm{OBDD}(\land, \mathrm{reordering})$ algorithm on the hard unsatisfiable formula, or it applies projection and then it has to simulate the running of an $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithm on the hard satisfiable formula.

Using this approach we can get a stronger result. Namely, we construct hard formulas for $1{-}\mathrm{NBP}(\land, \exists)$ algorithms that use non-deterministic read-once branching programs ($1{-}\mathrm{NBP}$) instead of OBDDs as the base representation model. $1{-}\mathrm{NBP}$ extends OBDD and is strictly more efficient. We should stress that $1{-}\mathrm{NBP}(\land, \exists)$ algorithms have no practical sense since for $1{-}\mathrm{NBP}$ we cannot efficiently compute the result of the conjunction with a clause. But a lower bound on the size of $1{-}\mathrm{NBP}$ trivially implies the same lower bound on the size of OBDD. We apply this extension in Theorem 20 to get that $1{-}\mathrm{NBP}(\land, \exists)$ algorithms and, thus, $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithms do not polynomially simulate tree-like resolution. This result extends the result of Segerlind that $\mathrm{OBDD}(\land, \exists)$ algorithms do not simulate dag-like resolution [20]. The separation formula can be obtained from the hard formula for $1{-}\mathrm{NBP}(\land, \exists)$ algorithms by adding several extension axioms. In Lemma 18 we show that adding extension axioms can not make the input formula simpler for $1{-}\mathrm{NBP}(\land, \exists)$ algorithm; the proof essentially exploits non-determinism in $1{-}\mathrm{NBPs}$. On the other hand, it is known that tree-like resolution with extension axioms is equivalent to the exceptionally strong proof system Extended Frege.

The disadvantage of the first family of hard formulas is that they are very artificial. So we prove the second lower bound for the formulas with a clear combinatorial meaning. The second family of hard formulas is the binary pigeonhole principle $\mathrm{BPHP}^{2^\ell+1}_{2^\ell}$ that encodes that there are $2^\ell + 1$ distinct binary strings of length ℓ. In Theorem 21 we show that every $\mathrm{OBDD}(\land, \exists, \mathrm{reordering})$ algorithm runs exponential time on $\mathrm{BPHP}^{2^\ell+1}_{2^\ell}$. The proof of the lower bound is rather technically involved.

2 Preliminaries

2.1 Branching programs

Let $X = \{x_1, \dots, x_n\}$ be a set of Boolean variables.

A *branching program* is a directed acyclic graph with one node with in-degree 0 and out-degree 2 (source), several inner nodes with out-degree 2, and two nodes with out-degree 0 (sinks). Every node except sinks is labeled with some variable from X, one of its outgoing edges is labeled with 0 and the other one is labeled with 1. One sink is labeled with 0 and the other with 1.

Each node v in a branching program computes a Boolean function f_v. The function f_v is defined recursively for v from sinks to the source. If v is a sink labeled with $k \in \{0,1\}$ then $f_v \equiv k$. Otherwise, suppose that v is labeled with a variable x_i its outgoing edge labeled with 0 goes to a node v_0 and its outgoing edge labeled with 1 goes to a node v_1. Then we define $f_v(x_1,\dots,x_n) = f_{v_0}(x_1,\dots,x_n)$ if $x_i = 0$ and $f_v(x_1,\dots,x_n) = f_{v_1}(x_1,\dots,x_n)$ if $x_i = 1$. Since the corresponding graph is acyclic the definition is correct for each node v. We say that a branching program computes the function that corresponds to its only source.

A branching program is called *read-once* (denoted 1-BP) if each its path contains at most one occurrence of each variable.

A branching program is called an *ordered binary decision diagram* (OBDD) if variables on every path from the source to sinks appear according to some fixed order of variables.

Sometimes we write π-OBDD instead of OBDD to emphasize that variables appear according to the order of variables π.

A *nondeterministic branching program* (NBP) is a directed acyclic graph with one node with in-degree 0 and out-degree 2 (source) several inner nodes with out-degree 2 and two nodes with out-degree 0 (sinks). Each node except sinks is either *common* and is labeled with some variable from X as in the definition of branching programs or *guessing* and is not labeled. Each *common* node has two outgoing edges. One of them is labeled with 0 and the other one is labeled with 1. Each *guessing* node has two non-labeled outgoing edges. One sink is labeled with 0 and the other with 1.

As in the definition of a branching program, each node v of NBP computes a Boolean function f_v. If v is a *common* node or a sink node then f_v is defined as in the case of deterministic branching programs. If v is a *guessing* node then it has two outgoing edges. Denote heads of the edges as $\{v_0, v_1\}$. Then we define $f_v = f_{v_0} \vee f_{v_1}$.

A NBP is called *read-once* (1-NBP) if each of its path contains at most one occurrence of each variable.

Let B be a deterministic or nondeterministic branching program. Then the size of B denoted by $|B|$ is the number of nodes in the corresponding graph.

Let $f(x_1,\dots,x_n)$ be a Boolean function and let B be a deterministic or nondeterministic branching program that computes f. Let ρ be a partial substitution into X. Then $f|_\rho = f(\rho(x_1),\dots,\rho(x_n))$ is a Boolean function that depends only on variables that are not in the ρ's domain. We denote by $B|_\rho$ the result of the following (syntactic) transformations of B: we perform the transformation from bottom to top. Suppose node v is labeled with a variable x_i such that $\rho(x_i) = a \in \{0,1\}$ and its outgoing edge labeled with 0 is going to a node v_0 and its other outgoing edge is going to v_1. Then we delete v from the graph and redirect edges that are going into v to the node v_a.

It is easy to see that $B|_\rho$ computes $f|_\rho$.

Also, note that the size of $B|_\rho$ is at most the size of the B.

▶ **Lemma 1** ([4], [22]).

1. *Let A and B be two π-OBDDs for some order of variables π. And let $\odot$ be a binary operation e.g $\wedge$, $\vee$, $\oplus$, etc. There exists an algorithm that takes A and B as inputs and returns a π-OBDD that computes $A \odot B$. Moreover, the algorithm runs in time $O(|A||B|)$. Therefore its output has size $O(|A||B|)$ (see Section 4 from [4]).*
2. *Let A be a π-OBDD and let x be its arbitrary variable. Then there exists an algorithm that takes A as an input and returns a π-OBDD that computes $\exists x A$ in time $O(|A|^2)$. Therefore the size of the output is also bounded by $O(|A|^2)$. (see Section 4 from [4])*
3. *Let A be a π_1-OBDD for some variables order π_1 and let π_2 be an another arbitrary variables order. There exists an algorithm that takes A as an input and returns a minimal size π_2-OBDD B such that $A \equiv B$ i.e. it computes the same Boolean function. Moreover, this algorithm runs at most* $\mathrm{poly}(|A| + |B|)$ *steps (see Section 5 from [22]).*

▶ **Lemma 2** (Lemma 4.2 from [11]). *Let D be a 1-NBP computing a Boolean function $f : \{0,1\}^n \to \{0,1\}$ and let $1 \le i \le n$. If we change every node in D labeled with the variable x_i by a guessing node and remove all labels of all its outgoing edges, then we obtain a valid 1-NBP that computes $\exists x_i f(x_1, x_2, \dots, x_n)$.*

2.2 Proof systems

A *resolution refutation* of an unsatisfiable CNF formula φ is a sequence of clauses $C_1, C_2, \dots, C_s$ such that (1) C_s is the empty clause (identically false), (2) for all $i \in [s]$, the clause C_i is either a clause of φ, or can be obtained by the resolution rule from two clauses with lesser numbers, where the resolution rule allows to derive $A \lor B$ from $A \lor x$ and $B \lor \neg x$. A resolution refutation is *tree-like* if every derived clause can be used as a premise of the resolution rule at most once.

Let $\varphi = \bigwedge_i C_i$ be an unsatisfiable CNF formula. An OBDD *refutation* [3, 14] of φ is a sequence of OBDDs $D_1, D_2, \dots, D_t$ such that D_t is the constant false OBDD and for all $1 \le i \le t$ the diagram D_i either represents a clause of φ or is obtained from the previous D_j's by one of the following derivation rules.

- **Conjunction (or join)** rule allows deriving a π-OBDD for $D_1 \land D_2$ from π-OBDDs D_1 and D_2. We emphasize here that the conjunction rule can be only applied to OBDDs with the same order of variables.
- **Projection** ($\exists$) rule allows deriving a π-OBDD represented $\exists x A$ from a π-OBDD represented A, where x a Boolean variable.
- **Weakening** rule allows deriving a π-OBDD represented B from a π-OBDD represented A if A semantically implies B, i.e. if every satisfying assignment of A also satisfies B.
- **Reordering** rule allows deriving an OBDD represented B from an OBDD represented A if A and B semantically equivalent (note that A and B may use different variable orders).

We consider several OBDD-based proof systems that use the rules defined above. We specify the allowed rules in the brackets e.g. the proof system OBDD($\land$) uses only the conjunction rule meanwhile the proof system OBDD($\land$, weakening) uses both the conjunction and the weakening rules, etc. All proof systems use the conjunction rule.

Note that the projection rule is a special case of the weakening rule, thus, both of them are usually not included in the brackets simultaneously.

The size of a refutation is the sum of the sizes of the OBDDs from it.

We also need to define 1-NBP semantical proof systems [5] that extends OBDD proof systems.

Let $\varphi = \bigwedge_i C_i$ be an unsatisfiable CNF formula. A 1-NBP refutation of φ is a sequence of 1-NBPs $D_1, D_2, \dots, D_t$ such that D_t is the constant false 1-NBP and for all $1 \le i \le t$ the diagram D_i either represents a clause of φ or obtained from the previous D_j's by one of the following derivation rules.

- **Conjunction (or join)** rule allows deriving a 1-NBP for $D_1 \land D_2$ from 1-NBPs D_1 and D_2.
- **Projection** ($\exists$) rule allows deriving a 1-NBP represented $\exists x A$ from a 1-NBP represented A where x is one of φ's boolean variables.
- **Weakening** rule allow deriving a 1-NBP represented B from a 1-NBP represented A if A semantically implies B.

As in the definition of OBDD proofs, 1-NBP proof systems can be defined with different sets of inference rules. For example $\mathrm{NBP}(\wedge, \exists)$ uses only the conjunction and the projection rules.

We emphasize that 1-NBP proof systems are not proof systems in the sense of Cook-Reckhow [8] unless P = NP since it is NP-hard to verify the correctness of a given proof.

Let $G(V, E)$ be a graph. Let $c\colon V \to \{0,1\}$ be a *charge function.* A *Tseitin formula* $\mathrm{T}(G, c)$ depends on the propositional variables x_e for $e \in E$. For each vertex $v \in V$ we define the parity condition of v as $P_v := \left(\sum_{e \ni v} x_e \equiv c(v) \bmod 2\right)$, where $e \ni v$ means that an edge e is incident to the vertex v. The Tseitin formula $\mathrm{T}(G, c)$ is the conjunction of parity conditions of all the vertices: $\bigwedge_{v \in V} P_v$. Tseitin formulas are represented in CNF as follows: we represent P_v in CNF in the canonical way for all $v \in V$.

▶ **Theorem 3** (Theorem 3.11 from [5]). *There exists a family of constant degree graphs G_n with n vertices such that any 1-NBP($\wedge$) refutation of an unsatisfiable Tseitin formula $\mathrm{T}(G_n, f)$ contains a 1-NBP of size at least $2^{\Omega(n)}$.*

2.3 OBDD algorithms for SAT

The algorithm gets as an input a CNF formula ϕ, it chooses some order π on the variables and creates both a π-ordered OBDD D (which initially is equal to the constant true function) and a set of clauses S (which initially consists of all clauses of the formula ϕ). While S is not empty the algorithm applies one of the following three operations:

- **Conjunction (or join)** delete some clause C from S and replace D by a π-OBDD that represents the conjunction $D \wedge C$;
- **Projection** ($\exists$) choose a variable x that has no occurrences in the clauses from S and replace D by a π-OBDD for the function $\exists x D$;
- **Reordering** choose a new order on variables π' and replace D by the equivalent π'-OBDD. Assign $\pi := \pi'$.

After every step of the algorithm, the following invariant is maintained: ϕ is satisfiable if and only if $\bigwedge\limits_{C \in S} C \wedge D$ is satisfiable. After the termination of the algorithm, the set S is empty; if the diagram D has a path from the source to a sink labeled by 1, then the algorithm returns "Satisfiable", otherwise it returns "Unsatisfiable".

We refer to the algorithms of this type as $\mathrm{OBDD}(\wedge, \exists, \text{reordering})$ algorithms. Besides, we use a similar notation for algorithms that use some of the operations: we just enumerate the used operations in the brackets. For example, the $\mathrm{OBDD}(\wedge)$ algorithms use only the conjunction operation, and the $\mathrm{OBDD}(\wedge, \exists)$ algorithms use only the conjunction and projection operations.

Since join and projection for OBDDs may be performed in polynomial time and reordering may be performed in time polynomial in the sizes of the input and the output, the running time of an $\mathrm{OBDD}(\wedge, \exists, \text{reordering})$ algorithm is polynomially related to the sum of the sizes of all states of the diagram D (here we ignore the time spent on choosing the next step of the algorithm).

We also define a purely theoretical notion of 1-NBP algorithms for SAT that naturally extends OBDD algorithms.

The algorithm gets as an input a CNF formula ϕ. It creates both a 1-NBP D (which initially is equal to the constant true function) and a set of clauses S (which initially consists of all clauses of the formula ϕ). While S is not empty the algorithm applies one of the following two operations:

- **Conjunction (or join)** delete some clause C from S and replace D by a 1-NBP that represents the conjunction $D \wedge C$;
- **Projection ($\exists$)** choose a variable x that has no occurrences in the clauses from S and replace D by a 1-NBP for the function $\exists x D$.

An execution track of a 1-NBP (OBDD) algorithm is the sequence of all 1-NBPs (OBDDs) constructed by the algorithm during its runtime. A *total size* of an execution track is the total size of 1-NBP (OBDDs) in it. By the running time of 1-NBP algorithms, we mean the sum of the sizes of all branching programs from its execution track. We will mainly use the lower bound of the running time of 1-NBP algorithms as a lower bound for OBDD algorithms.

▶ **Lemma 4.** *From every execution track of an* OBDD($\wedge, \exists$, *reordering*) *algorithm one can remove several diagrams to get an execution track of a correct* $1-$NBP($\wedge, \exists$) *algorithm.*

Proof. Since OBDD is a special case of $1-$NBP, applications of conjunction and join operations of OBDD($\wedge, \exists$, reordering) algorithm are legal operations for 1-NBP($\wedge, \exists$) algorithms. Notice that the reordering rule does not change the Boolean function, so we can just remove the larger of two OBDDs representing the same Boolean function. ◀

2.4 Quantified Boolean formulas

An $\exists$-CNF formula is a formula of type $\exists x_{i_1} \exists x_{i_2}, \ldots \exists x_{i_k} \phi(x_1, x_2, \ldots, x_n)$, where $\phi(x_1, x_2, \ldots, x_n)$ is a CNF formula, $\{i_1, i_2, \ldots, i_k\} \subseteq [n]$ and k is non-negative integer. The formula ϕ is called *the matrix* of the $\exists$-CNF formula.

▶ **Lemma 5.** *Let φ be an $\exists$-CNF formula. Assume that a partial assignment ρ satisfies φ. Then for every variable x, ρ satisfies $\exists x \varphi$.*

Proof. If x does not have occurrences in φ, then $\exists x \varphi$ is equivalent to φ and, therefore, it is satisfied by ρ. Otherwise $\exists x \varphi$ is semantically equivalent to $\varphi|_{x:=\rho(x)} \vee \varphi|_{x:=1-\rho(x)}$. By trivial reasons ρ satisfies $\varphi|_{x:=\rho(x)}$, hence it satisfies $\exists x \varphi$. ◀

2.5 Error-correcting codes

We will use error-correcting codes to construct hard formulas and analyze the running time of the OBDD algorithms.

By a *code* we mean a subset of binary strings with a fixed length. A code C has a distance d if for any two codewords $c_1, c_2 \in C$ the Hamming distance between c_1 and c_2 is at least d. A code $C \subseteq \{0,1\}^n$ has a relative distance δ if it has the distance δn.

A *linear code* is a set of all n-bits vectors $x = (x_1 \ldots x_n)$ from some linear subspace in $\mathbb{F}_2^n$.

A linear code can be specified by a system of linear equations. For a code of dimension k this system should consist of $m \geq n - k$ linear equations involving n variables. The set of all solutions of the system should give exactly our code, so the rank of the system must be equal to $n - k$. If we require in addition that the equations in the system are linearly independent, then the number of equations is equal to $m = n - k$. The matrix of this linear system is called a *checksum matrix* of the code.

▶ **Lemma 6** (Hamming code [13])**.** *There is a linear code $C \subseteq \{0,1\}^n$ of size $2^{\Omega(n/\log n)}$ with distance* 3.

For $0 < p < 1$ *the binary entropy* is $H(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$. We will use the binary entropy to estimate the size of a linear code with the given relative distance.

▶ **Lemma 7** (Gilbert-Varshamov Bound [9], [21]). *Let $0 < \varepsilon < \frac{1}{2}$. Then there exists a linear code $C \subseteq \{0,1\}^n$ of size at least $2^{(1-H(\varepsilon))n}$ with relative distance ε.*

2.6 Communication complexity

Communication complexity is one of the ways to estimate the size of OBDD representation of Boolean functions.

Let $f : A \times B \to C$ be a function. Two players Alice and Bob want to compute $f(a,b)$ for some $a \in A$ and $b \in B$. However, Alice knows only a and Bob knows only b. In order to compute the value they can use a two-sided communication channel. They agreed in advance on a protocol; at each step of the protocol, one of them sends a bit string to the other, at the end of the protocol both Alice and Bob should know $f(x)$. The cost of the protocol is the maximal number of bits they sent to each other. The communication complexity of f is equal to the minimal cost of the protocols for f (for the formal definition see [16]).

Let $f : A \times B \to C$ be a function. A set $S \subseteq A \times B$ is called a *fooling set* if exists $z \in C$ such that for all $(a,b) \in S$, $f(a,b) = z$. But for all $a_1 \neq a_2 \in A$ and $b_1 \neq b_2 \in B$ if $(a_1,b_1) \in S$ and $(a_2,b_2) \in S$, then $f(a_1,b_2) \neq z$ or $f(a_2,b_1) \neq z$.

▶ **Lemma 8** (Lemma 1.20 from [16]). *If S is a fooling set of size k for a function $f : A \times B \to C$. Then the communication complexity of f with respect to partition (A,B) is at least $\log k$.*

▶ **Lemma 9** (Lemma 12.12 from [16]). *Let $f : A \times B \to \{0,1\}$ be a Boolean function. Assume that t is the deterministic communication complexity of f with respect to partition A and B. Then for every variable order π that respects this partition (i.e. all variables from A are π-less then all variables from B or vice versa), the size of any $\pi-$OBDD computing f is at least 2^t.*

3 Lower bounds for $1-\mathrm{NBP}$ algorithms

In this section, we give a construction of a hard unsatisfiable formula for $1-\mathrm{NBP}(\wedge,\exists)$ algorithms. In Subsection 3.1 we show how to get it from a hard satisfiable formula for $1-\mathrm{NBP}(\wedge,\exists)$ algorithms and a hard formula for $1-\mathrm{NBP}(\wedge)$ proofs. In Subsection 3.2 we show that satisfiable formulas from [14] that are hard for $\mathrm{OBDD}(\wedge,\exists,\mathrm{reordering})$ algorithms are also hard for $1-\mathrm{NBP}(\wedge,\exists)$ algorithms. In Subsection 3.3 we show that $1-\mathrm{NBP}(\wedge,\exists)$ algorithms do not polynomially simulate tree-like resolution.

3.1 Hard unsatisfiable formula

Let Φ be a CNF formula and let x be a Boolean variable that does not appear in Φ. We denote by $x \vee \Phi$ the CNF formula that is obtained from Φ by adding x to each of its clauses.

Let Φ and Ψ be CNF formulas in m and n Boolean variables respectively. We define a formula $\mathcal{F}(\Phi,\Psi)$ in $n + 2mn$ variables $X = \{x_i, y_j^{(i)}, z_j^{(i)} \mid i \in [n], j \in [m]\}$ as follows:

$$\mathcal{F}(\Phi,\Psi) := \Psi(x_1,\dots,x_n) \wedge T(X),$$

where

$$T(X) := \bigwedge_{i=1}^{n} \left(x_i \vee \Phi\left(y_1^{(i)}, y_2^{(i)}, \dots, y_m^{(i)}\right)\right) \wedge \bigwedge_{i=1}^{n} \left(\neg x_i \vee \Phi\left(z_1^{(i)}, z_2^{(i)}, \dots, z_m^{(i)}\right)\right).$$

Note that if Ψ is unsatisfiable then so is $\mathcal{F}(\Phi,\Psi)$.

▶ **Theorem 10.** *Let Φ be a satisfiable CNF formula in n variables and let Ψ be an unsatisfiable CNF formula in m variables. Suppose that the execution track of every $1{-}\mathrm{NBP}(\wedge,\exists)$ algorithm on the input $\Phi(y_1,\dots,y_m)$ contains a $1{-}\mathrm{NBP}$ of size at least S_1 and every 1-NBP($\wedge$) refutations of $\Psi(x_1,\dots,x_n)$ contains a $1{-}\mathrm{NBP}$ of size at least S_2 and $S_2 > n+1$. Then any execution track of a 1-NBP($\wedge,\exists$) algorithm on the input $\mathcal{F}(\Phi,\Psi)$ contains a $1{-}\mathrm{NBP}$ of size at least $\min(S_1,S_2)$.*

Proof. Consider an 1-NBP($\wedge,\exists$) algorithm. Let $B_1, B_2, \dots, B_\ell$ be its execution track on the input $\mathcal{F}(\Phi,\Psi)$. For all $i \in [\ell]$, B_i represents a $\exists$-CNF formula whose matrix is the conjunction of a subset of clauses of the input formula; we denote this $\exists$-CNF formula by F_i.

We consider two cases of what happened earlier: the matrix of F_j contains an unsatisfiable set of clauses from $\Psi(x_1,\dots,x_n)$, or F_j is quantified over some variable x_i for $i \in [n]$.

Case 1. There exists $k \in [\ell]$ such that the matrix of F_k contains an unsatisfiable set of clauses from $\Psi(x_1,\dots,x_n)$ and F_k itself is not quantified over x_i for all $i \in [n]$.

Since the formula Φ is satisfiable, there exist an assignment ρ of the variables $y_i^{(j)}, z_i^{(j)}$ for $1 \le i \le n$, $1 \le j \le m$ such that all copies of Φ in $\mathcal{F}(\Phi,\Psi)$ are satisfied by ρ.

Consider the sequence $B_1|_\rho, \dots, B_k|_\rho$. For every $i \in [k-1]$:

- if $B_{i+1} \equiv B_i \wedge C$, where C is a clause of Ψ, then $B_{i+1}|_\rho \equiv B_i|_\rho \wedge C$; here and after $\equiv$ means the semantical equivalence of Boolean functions;
- if $B_{i+1} \equiv B_i \wedge C$, where C is a clause of $T(X)$, then $B_{i+1}|_\rho \equiv B_i|_\rho$;
- if $B_{i+1} \equiv \exists z B_i$, where $z \in X \setminus \{x_1, x_2, \dots, x_n\}$, then $B_{i+1}|_\rho = B_i|_\rho$;

Let $\tilde{C}_1, \tilde{C}_2, \dots, \tilde{C}_s$ be $1{-}\mathrm{NBP}$ representations of all clauses of Ψ. It is easy to see that $|\tilde{C}_i| \le n+1$ for all $i \in [s]$. Then $\tilde{C}_1, \tilde{C}_2, \dots, \tilde{C}_s, B_1|_\rho, \dots, B_k|_\rho$ is the correct 1-NBP($\wedge$) refutation of Ψ. By the properties of the formula Ψ, any refutation should contain a $1{-}\mathrm{NBP}$ of size at least S_2. Since $S_2 > n+1$, there is $j \in [k]$ such that $|B_j|_\rho|$ is at least S_2, hence, $|B_j| \ge S_2$.

Case 2. Let p be the minimal number such that F_p is quantified with some variable x_i for $i \in [n]$ (we denote this variable by x_{i_0}). In the considered case, the matrix of F_p does not contain an unsatisfiable set of clauses from Ψ. Notice that $F_p = \exists x_{i_0} F_{p-1}$.

For $j \in [p]$, F_j is equivalent to $\Psi'_j \wedge \Theta_j$, where Ψ'_j is the satisfiable conjunction of several clauses of Ψ and Θ_j is the $\exists$-CNF formula whose matrix is the conjunction of several clauses from $T(X)$. Let $\alpha : \{x_1, x_2, \dots, x_n\} \to \{0,1\}$ be a satisfying assignment of Ψ'_{p-1}. Notice that α satisfies all Ψ'_j for $j \in [p-1]$. Let $\beta : \{t_1, t_2, \dots, t_m\} \to \{0,1\}$ be a satisfying assignment of $\Phi(t_1, t_2, \dots, t_m)$.

Let us define a partial assignment ρ to the variables X.

- $\rho(x_i) = \alpha(x_i)$, for $i \in [n]$;
- $\rho(z_i^{(j)}) = \beta(t_i)$, for $i \in [m], j \in [n] \setminus \{i_0\}$;
- $\rho(y_i^{(j)}) = \beta(t_i)$, for $i \in [m], j \in [n] \setminus \{i_0\}$.

▷ Claim 11. It is possible to delete several $1{-}\mathrm{NBP}$s from the sequence $B_1|_\rho, \dots, B_{p-1}|_\rho$ to get a correct execution track of 1-NBP($\wedge,\exists$) algorithm executed on $\Phi\left(y_1^{(i_0)}, \dots, y_m^{(i_0)}\right)$ if $\alpha(x_{i_0}) = 0$ and on $\Phi\left(z_1^{(i_0)}, \dots, z_m^{(i_0)}\right)$, otherwise.

Claim 11 and the property of the formula Φ imply that there exists $i \in [p-1]$ such that $|B_i|_\rho| \ge S_1$, therefore, $|B_i| \ge S_1$.

Proof of Claim 11. W.l.o.g. assume that $\alpha(x_{i_0}) = 0$. Since α satisfies Ψ'_{p-1}, α satisfies Ψ'_j for all $j \in [p-1]$. Hence ρ satisfies Ψ'_j for all $j \in [p-1]$.

Let us represent $\Theta_j = \bigwedge_{i=1}^n U_j^{(i)} \wedge H_j^{(i)}$, where $U_j^{(i)}$ is $\exists$-CNF formula whose matrix is the conjunction of several clauses from $\left(x_i \vee \Phi\left(y_1^{(i)}, \ldots, y_m^{(i)}\right)\right)$ and $H_j^{(i)}$ $\exists$-CNF formula whose matrix is the conjunction of several clauses from $(\neg x_i \vee \Phi(z_1^{(i)}, \ldots, z_m^{(i)}))$. For all $i \neq i_0$, the matrices of $U_j^{(i)}$ and $H_j^{(i)}$ are satisfied by ρ, hence by Lemma 5, $U_j^{(i)}$ and $H_j^{(i)}$ themselves are satisfied by ρ.

Since $\alpha(x_{i_0}) = 0$, ρ satisfies $H_j^{(i_0)}$. Hence, $F_j|_\rho = U_j^{(i_0)}$ and the matrix of $U_j^{(i_0)}$ is the conjunction of several clauses of $\Phi\left(y_1^{(i_0)}, \ldots, y_m^{(i_0)}\right)$. Since F_p is quantified over x_{i_0}, all clauses from $\mathcal{F}(\Phi, \Psi)$ containing x_{i_0} should be in the matrix of F_{p-1}, hence the matrix of $F_{p-1}|_\rho$ is exactly $\Phi\left(y_1^{(i_0)}, \ldots, y_m^{(i_0)}\right)$.

For every $j \in [p-2]$,

- if $B_{j+1} \equiv B_j \wedge C$ and $C = x_{i_0} \vee C'$, where C' is a clause of $\Phi\left(y_1^{(i_0)}, \ldots, y_m^{(i_0)}\right)$, then $B_{j+1}|_\rho \equiv B_j \wedge C'$;
- if $B_{j+1} \equiv \exists y_k^{(i_0)} B_j$ and $k \in [m]$, then $B_{j+1}|_\rho \equiv \exists y_k^{(i_0)} B_j|_\rho$.
- In all other cases $B_{j+1}|_\rho \equiv B_j|_\rho$. For such j, B_{j+1} will be deleted from the sequence as required by the claim. ◁

◀

The following lemma is proved in Subsection 3.2.

▶ **Lemma 12** (cf. Corollary 5.4 from [14]). *For all large enough n there exists a satisfiable CNF formula with n Boolean variables, of size $O(n)$ such that the execution track of every* $1-\mathrm{NBP}(\wedge, \exists)$ *algorithm running on this formula contains a* $1-\mathrm{NBP}$ *of size at least $2^{\Omega(n)}$. Moreover, for a given n such a formula can be constructed by a deterministic algorithm in time* $\mathrm{poly}(n)$.

▶ **Corollary 13.** *In poly(n) time one can construct an unsatisfiable formula F_n in* $\mathrm{poly}(n)$ *variables such that the execution track of every* $1-\mathrm{NBP}(\wedge, \exists)$ *algorithm running on the formula F_n contains* $1-\mathrm{NBP}$ *of size at least $2^{\Omega(n)}$.*

Proof. Let Ψ_n be a Tseitin formula $T(G_n, f)$ based on the graph G_n from Theorem 3. By Theorem 3, any $1-\mathrm{NBP}(\wedge)$ refutation of Ψ_n contains a $1-\mathrm{NBP}$ of size at least $2^{\Omega(n)}$. Let Φ_n be a satisfiable formula from Lemma 12. Then we can take $F_n = \mathcal{F}(\Psi, \Phi)$ and it has the required property by Theorem 10. ◀

3.2 Hard satisfiable formulas for $1-\mathrm{NBP}(\wedge, \exists)$ algorithms

In this section, we prove Lemma 12. The proof is mainly repeating the proof from [14] for the case of $\mathrm{OBDD}(\wedge, \exists, \mathrm{reordering})$ algorithms.

We say that a code $C \subseteq \{0,1\}^n$ recovers a ρ fraction of erasures by a list of size L (or C is (ρ, L)-erasure list-decodable) if for any $w \in \{0, 1, ?\}^n$ such that the number of ? in w does not exceed ρn, there exist at most L elements in C that are consistent with w. A string $s \in \{0,1\}^n$ is consistent with w if for all i, $w_i \in \{0,1\}$ implies $s_i = w_i$.

▶ **Theorem 14** ([12, Lemma 2]). *If $\mathcal{C}$ is a code with relative distance δ, then for every $\epsilon > 0$ the code C is $((2-\epsilon)\delta, \frac{2}{\epsilon})$-erasure list-decodable.*

The following theorem states that every $1-\mathrm{NBP}$ for a characteristic function of a good enough code has at least exponential size. It extends Theorem 5.2 from [14] which claims the same lower bound on the size of OBDDs.

▶ **Theorem 15** (cf. Theorem 5.2 from [14]). *Let $C \subseteq \{0,1\}^n$ be a $(\frac{1}{2}+\epsilon, L)$-erasure list-decodable code with relative distance more than 2ϵ. Any $1-$NBP representation of the characteristic function of C (i.e. function $\chi_C \colon \{0,1\}^n \to \{0,1\} : \forall x \in \{0,1\}^n\ \chi_C(x) = 1 \Leftrightarrow x \in C$) has size at least $\frac{|C|}{L^2}$.*

Moreover, for every tuple of k different indices $i_1, \dots, i_k \in [n]$ $(0 \le k \le 2\epsilon n)$ size of any $1-$NBP representation of the Boolean function $\exists x_{i_1} \dots \exists x_{i_k}\ \chi_C(x_1, \dots, x_n)$ is at least $\frac{|C|}{L^2}$.

Proof. It is enough to prove the "moreover" part of the statement since the first part is its special case (with $k = 0$).

Notice that since any string of size $\lceil(\frac{1}{2}-\epsilon)n\rceil$ has at most L prolongation to an element of C, $|C| \le 2^{\lceil(\frac{1}{2}-\epsilon)n\rceil}L$. We may assume that $L < 2^{\lfloor(\frac{1}{2}-\epsilon)n\rfloor}$, since otherwise $\frac{|C|}{L^2} \le 2$ and the theorem is trivial.

Let D be a $1-$NBP computing $\exists x_{i_1} \dots \exists x_{i_k}\ \chi_C(x_1, \dots, x_n)$.

Consider $|C|$ codewords, for each of them there is a path in D from the source to 1-sink that is consistence with the codeword. For each such path, we mark a node v such that between the source and v there are queried exactly $\lceil\frac{n-k}{2}\rceil$ variables (the query in v is not included). We claim that such vertex always exists. Indeed, assume for the sake of contradiction that there is an accepting path p corresponding to a codeword $c \in C$ that queries t variables and $t < \lceil\frac{n-k}{2}\rceil$ variables. There are at least $2^{\lfloor\frac{n-k}{2}\rfloor}$ partial assignments from $\{x_i \mid i \in [n] \setminus \{i_1, i_2, \dots, i_k\}\} \to \{0,1\}$ such that each of them is consistent with c in the values of at least $\lceil\frac{n-k}{2}\rceil$ positions including all variables from the path p. Since all these assignments are consistent with p, they are accepted by D, hence each of them may be continued to a codeword. Hence there are at least $2^{\lfloor\frac{n-k}{2}\rfloor}$ codewords that agree with c in at least $\lceil\frac{n-k}{2}\rceil$ positions. Hence, $2^{\lfloor(\frac{1}{2}-\epsilon)n\rfloor} \le 2^{\lfloor\frac{n-k}{2}\rfloor} \le L$. This contradicts our assumptions on the size of L.

Let us estimate from the above the number of times that the same node can be marked. We claim that every marked node v can be marked at most L^2 times. Assume for the sake of contradiction that there is a node v that is marked at least L^2+1 times. Let $S \subseteq C$ be a set of codewords such that the node v was marked on the paths corresponding to them. Consider a codeword $s \in S$ and the path which is consistence with s, let between the source and v (not including the query in v) the set of queried variables be equal to $\{x_i \mid i \in I\}$. Let $J = ([n] \setminus \{i_1, i_2, \dots, i_k\}) \setminus I$. Since the relative distance of C is more than 2ϵ, no two codewords coincide on $([n] \setminus \{i_1, i_2, \dots, i_k\})$. Hence, by the pigeonhole principle, the set S contains at least $L+1$ elements with different projections on the set I, or at least $L+1$ elements with different projections on J. Consider these cases separately.

Assume that S contains $L+1$ elements with different projections on I: $s, s_1, \dots, s_L$. Consider partial assignments $\tau_1, \tau_2, \dots, \tau_L : \{x_i \mid i \in I \cup J\} \to \{0,1\}$, where for all $j \in [L]$ and for all $i \in I$, $\tau_j(x_i)$ equals the ith bit of s_j and for all $i \in J$ and $\tau_j(x_i)$ equals the ith bit of s. Since s is a codeword, it is accepted by D; for all $i \in [L]$, τ_i is also accepted by D by the following accepting path: from the source to v we follow the path corresponding s_i and from v to 1-sink we follow the path corresponding s. Hence for every $j \in [\ell]$, τ_j can be extended to a codeword t_j. By the construction $s, \tau_1, \dots, \tau_L$ coincide in the set of positions J, i.e., in $\frac{n-k}{2} \ge (\frac{1}{2}-\epsilon)n$ bits, hence there exists $L+1$ different codewords that coincide in $(\frac{1}{2}-\epsilon)n$ positions, this contradicts the property of the code.

In the second case, S contains $L+1$ elements with different projections on J: $s, s_1, \dots, s_\ell$. This case can be handled analogously to the previous one. The only difference is the definition of τ_j for $j \in [\ell]$. For $i \in I$, $\tau(x_i)$ equals the ith bit of s and for $j \in I$, $\tau(x_i)$ equals the ith bit of s_j.

So we get that there are at least $\frac{|C|}{L^2}$ distinct marked nodes in D. ◀

Using Theorem 15 we extend Theorem 5.3 from [14] from OBDD($\wedge, \exists$, reordering) algorithms to 1−NBP($\wedge, \exists$) algorithms.

▶ **Theorem 16** (cf. Theorem 5.3 from [14]). *Let $C \subseteq \{0,1\}^n$ be a linear code with the relative distance $\frac{1}{3}$ such that the checksum matrix H of the code C has the following properties:*

- *H is a binary matrix of size $\alpha n \times n$, where $\alpha \in (0,1)$ is a constant;*
- *every row of H contains at most $t(n)$ ones, where t is some function;*
- *in every $\frac{1}{6}n$ columns of H we can find ones in at least $(\alpha - \delta)n$ different rows of the matrix, where $\delta \in \left(0, \min\{\alpha, \frac{1-\alpha}{2}\}\right)$ is a constant.*

Denote by F_n the canonical CNF representation of the system of linear equations $H(x) = 0$; F_n is in $t(n)$-CNF, hence the size of F_n is at most $\alpha n 2^{t(n)-1} t(n)$. Then the execution track of every 1−NBP($\wedge, \exists$) *algorithm running on F_n contains a* 1−NBP *of size at least $2^{\Omega(n)}$.*

Proof. The proof resembles the proof of Theorem 5.3 from [14]. The only source of OBDD size lower bound in that proof is the usage of Theorem 5.2 from [14]; we will use Theorem 15 instead to get a lower bound on the size of 1−NBP. ◀

Lemma 12 follows from Theorem 16 applied to the codes constructed in [14].

3.3 Comparison with tree-like Resolution

▶ **Definition 17.** *Let $\mathcal{P}$ be a proof system used to derive refutations of CNF formulas. A set E of extension axioms for a set of propositional variables $\vec{x}$ is a set of clauses expressing $z_i := \psi_i(\vec{x}, z_1, \ldots, z_{i-1})$, where each ψ_i is a conjunction of literals and $z_1, \ldots, z_\ell$ are new variables. Let $\varphi(\vec{x})$ be a CNF formula. Then, an* extension-$\mathcal{P}$ *refutation of $\varphi(\vec{x})$ is by definition a refutation of $\varphi \wedge E$ where E is a set of extension axioms for $\vec{x}$.*

For example, a definition by extension of the form $z := (y_1 \wedge y_2)$ is represented by the three clauses $\neg z \vee y_1$, $\neg z \vee y_2$, and $\neg y_1 \vee \neg y_2 \vee z$.

▶ **Lemma 18.** *Let φ be a CNF formula and let E be a set of extension rules represented in CNF. Then for any execution track of* 1−NBP($\wedge, \exists$) *algorithm on the input $\varphi \wedge E$ exists a correct execution track of* 1−NBP($\wedge, \exists$) *algorithm on the input φ of not greater total size.*

Proof. It is sufficient to show it for the case when E contains just one extension rule. We assume that E contains the only rule $z := \psi(y_1, \ldots, y_k)$, where y_i are variables of φ. Let $D_1, D_2, \ldots, D_s$ be an execution track of an 1−NBP($\wedge, \exists$) algorithm on the formula $\varphi \wedge E$.

Every D_i represent a $\exists$-CNF formula with a matrix $(\Phi \wedge \Psi)$, where Φ is a conjunction of clauses of φ, and Ψ is a conjunction of several clauses from E.

Notice that since E is a CNF representation of $z := \psi(y_1, \ldots, y_k)$, then $\exists z \Psi$ is identically true. Since z has no occurrences in φ, $\exists z D_i$ is equivalent to $\exists x_{i_1} \ldots \exists x_{i_\ell} \Phi$.

By Lemma 2 there exists a 1−NBP representing $\exists D_i$ of size at most the size of D_i. It is easy to see that $\exists z D_1, \exists z D_2, \ldots, \exists z D_s$ may be considered (possibly after deletion of several extra 1−NBPs) as a correct execution track of a 1−NBP($\wedge, \exists$) algorithm on the formula φ. ◀

▶ **Lemma 19** (Lemma 4.6 from [5]). *Let G_n be an undirected graph with n vertices, with all vertices of degree at most d. Let f_n be a labeling function for G_n such that the Tseitin formula $\mathrm{T}(G_n, f_n)$ is unsatisfiable. Then there is a set E of extension axioms for $\mathrm{T}(G_n, f_n)$ of size* poly(n) *such that there is a tree-like resolution refutation of $\mathrm{T}(G_n, f_n) \wedge E$ of size* poly(n).

▶ **Theorem 20.** *There is a family of CNF formulas φ_n of size $\mathrm{poly}(n)$ such that φ_n has poly(n) tree-like resolution refutation but any execution track of $1\text{−NBP}(\wedge, \exists)$ algorithm on φ_n contains a 1−NBP of size $2^{\Omega(n)}$.*

Proof. Consider a hard formula F_n from Corollary 13. F_n has a form $\mathrm{T}(G_n, f_n) \wedge \psi$, where ψ is a satisfiable formula. By Lemma 19, there exists a set E of extension axioms for $\mathrm{T}(G_n, f_n)$ of size $\mathrm{poly}(n)$ such that there is a tree-like resolution refutation of $\mathrm{T}(G_n, f_n) \wedge E$ of size $\mathrm{poly}(n)$. Notice that $\mathrm{T}(G_n, f_n) \wedge E \wedge \psi$ also has a tree-like resolution refutation of size $\mathrm{poly}(n)$. By Corollary 13 and Lemma 18, the execution track of any $1\text{−NBP}(\wedge, \exists)$ algorithm on the formula $\mathrm{T}(G_n, f_n) \wedge E \wedge \psi$ contains a 1−NBP of size $2^{\Omega(n)}$. ◀

4 Binary pigeonhole principle is hard for $\mathrm{OBDD}(\wedge, \exists, \text{reordering})$ algorithms

The Binary Pigeonhole Principle represents in CNF that it is impossible to have $2^l + 1$ distinct binary strings of length l. Let $X = \{X_{i,j} \mid i \in [2^l + 1], j \in [l]\}$ be a set of propositional variables. For every $i \in [2^l + 1]$ we denote the vector of variables $(X_{i,1}, \ldots, X_{i,l})$ as X_i. For all distinct i and j from $[2^l + 1]$ and for all binary strings $a \in \{0, 1\}^l$ we define a clause $C^a_{i,j}$ that encodes that at least one of the strings X_i or X_j differs from a as follows $C^a_{i,j} = \bigvee\limits_{m=1}^{l} (X_{i,m} \neq a_m \vee X_{j,m} \neq a_m)$, where for a propositional variable x, $x \neq 0$ denotes x and $x \neq 1$ denotes $\neg x$. Let us denote $[X_i \neq X_j] := \bigwedge\limits_{a\in\{0,1\}^l} C^a_{i,j}$ the CNF formula that encodes that $X_i \neq X_j$. We finally define $\mathrm{BPHP}^{2^l+1}_{2^l} := \bigwedge\limits_{i,j\in[2^l+1]:i\neq j} [X_i \neq X_j]$.

It is convenient to consider X as a $(2^l + 1) \times l$ matrix of propositional variables. In this paper, we refer to X as *the variables matrix* or simply *the matrix* and to X_i for $i \in [2^l + 1]$ as the ith row of the matrix.

The main result of this section is the following.

▶ **Theorem 21.** *Let $n = 2^l$ and $0 < \varepsilon < 1$ be a solution of the equation $\varepsilon = 1 - H(\varepsilon)$ ($\varepsilon \approx 0.227092$). Then the execution track of any $\mathrm{OBDD}(\wedge, \exists, \textit{reordering})$ algorithm on the input BPHP^{n+1}_n contains an OBDD of size at least $2^{\Omega(n^\varepsilon/\log n)}$.*

Proof. Consider an $\mathrm{OBDD}(\wedge, \exists, \text{reordering})$ algorithm $\mathcal{A}$ and its execution on the input BPHP^{n+1}_n.

We consider several cases:

Case 1. Suppose that during its running the algorithm $\mathcal{A}$ applied projections to variables that all lie in at most $n/2$ different rows of the variables matrix X. Consider the state of the diagram D before the last application of the conjunction operation. In this moment D represents an $\exists$-CNF formula whose matrix is the conjunction of all but one clauses of BPHP^{n+1}_n. The following lemma implies that in this case $|D| \geq 2^{\Omega(n/\log n)}$.

▶ **Lemma 22.** *Let $\mathcal{C}$ be the strict subset of the set of clauses of BPHP^{n+1}_n. Let $Z = \{z_1, \ldots, z_b\} \subseteq X$ be a set of BPHP^{n+1}_n variables. Assume that there is a set $Y \subseteq [n+1]$ such that*

1. *for all $i \in Y$, X_i does not contain variables from Z;*
2. *$|Y| \geq n/\log n$;*
3. *for all $i, j \in Y$, $\mathcal{C}$ contains all clauses representing $[X_i \neq X_j]$.*

Then any OBDD representing $\varphi := \exists z_1, \ldots, z_b \bigwedge_{C\in\mathcal{C}} C$ has size at least $2^{\Omega(n/\log n)}$.

We prove Lemma 22 in Subsection 4.1.

Case 2. The algorithm $\mathcal{A}$ applied projection operation over variables from more than $n/2$ rows of the variables matrix X. Since $\mathcal{A}$ applies projection operations one by one, one of the following events happens before the other:

1. There is a row of the variable's matrix X such that there are $\lfloor \varepsilon l \rfloor - 3$ variables in this row for which $\mathcal{A}$ applied projection operations.
2. $\mathcal{A}$ applies projections over variables from $\lfloor n^{\varepsilon}/100 \log n \rfloor$ rows of the variables matrix X.

Case 2.1. Assume that the first event happened before the second. Consider the state of the diagram D right after $\mathcal{A}$ first time has applied projection over $\lfloor \varepsilon l \rfloor - 3$ variables from some row. Since $\varepsilon < 1/4$, the next lemma implies that $|D| \geq 2^{\Omega(n^{\varepsilon}/\log n)}$.

▶ **Lemma 23.** *Let $\mathcal{C}$ be a proper subset of clauses from BPHP_n^{n+1}. Let $Z = \{z_1, \dots, z_b\} \subseteq X$ be a set of BPHP_n^{n+1} variables. Suppose that for all $z \in Z$ all occurrences of variables of Z in BPHP_n^{n+1} are in clauses from $\mathcal{C}$. Suppose also that:*

1. *There exists a row of X containing exactly k variables from Z.*
2. *Variables from Z occur in at most $n/2$ rows of X.*

Then every OBDD that computes $\varphi := \exists z_1, \dots, z_b \bigwedge_{C \in \mathcal{C}} C$ has size at least $\min\left\{2^{\Omega(2^k/k)}, 2^{\Omega\left(2^{l-3k}/l^2\right)}\right\}$.

We prove Lemma 23 in Subsection 4.2.

Case 2.2. Now assume that the second event happened before the first. Consider the state of the diagram D right after $\mathcal{A}$ first time has applied projection over variables from $\lfloor n^{\varepsilon}/100 \log n \rfloor$ different rows.

▶ **Lemma 24.** *Let $\mathcal{C}$ be a strict subset of the set of clauses of BPHP_n^{n+1}. Let $Z = \{z_1, \dots, z_b\} \subseteq X$ be a set of BPHP_n^{n+1} variables. Suppose that for all $z \in Z$ all clauses BPHP_n^{n+1} having variables from Z are in $\mathcal{C}$. Let $d \in [l-1]$ and $T \in [2^l]$ be such that*

1. *variables from Z occur in exactly T rows of the variables matrix X;*
2. *each row of X contains at most d variables from Z;*
3. *there exists a code $ECC \subseteq \{0,1\}^l$ with distance $d+3$ and size $|ECC| = T+1$;*
4. *$1 + (l+1) \cdot T \cdot 2^{d+1} < n/2$.*

Then every OBDD that computes $\varphi := \exists z_1, \dots, z_b \bigwedge_{C \in \mathcal{C}} C$ has size at least 2^T.

The proof of Lemma 24 is given in Subsection 4.3.

By Lemma 7, there exists a code $ECC \subseteq \{0,1\}^l$ with distance $\varepsilon\ell$ and size $2^{(1-H(\varepsilon))\ell} = n^{1-H(\varepsilon)} = n^{\varepsilon}$, hence there exists code with the same distance of size $T := \lfloor n^{\varepsilon}/(100 \log n) \rfloor + 1$. It is straightforward that $1 + (l+1) \cdot T \cdot 2^{d+1} < n/2$. Hence, Lemma 24 implies that $|D| \geq 2^{\Omega(n^{\varepsilon}/\log n)}$. ◀

4.1 Proof of Lemma 22

▶ **Lemma 25.** *Let $\mathcal{C}$ be a subset of clauses from BPHP_n^{n+1}, where $n = 2^l$. Assume that $i_0, j_0 \in [n+1]$ and $a_0 \in \{0,1\}^{\ell}$ are such that the clause $(X_{i_0} \neq a_0 \vee X_{j_0} \neq a_0)$ does not belong to $\mathcal{C}$. Let $Z = \{z_1, \dots, z_b\} \subseteq X$ be a set of variables of BPHP_n^{n+1}. Let the set of variables X be partitioned into two parts $X^{(1)}$ and $X^{(2)}$. There are $2t$ rows of the variables matrix: $s_1^1, s_1^2, s_2^1, s_2^2 \dots, s_t^1, s_t^2 \in [n+1] \setminus \{i_0, j_0\}$ such that*

1. *for all $i \in [t]$, the rows s_i^1 and s_i^2 does not contain variables from Z.*
2. *for all $i \in [t]$, the row s_i^1 contains at least one variable from the first part $X^{(1)}$ and the row s_i^2 contains only variables from the second part $X^{(2)}$;*

3. *There are t binary strings $a_1, a_2, \dots, a_t \in \{0,1\}^\ell \setminus \{a_0\}$ such that the Hamming distance between a_i and a_j is at least 3 and for all $i \in \{0, 1, \dots, t\}$ the clause $(X_{s_i^1} \neq a_i \vee X_{s_i^2} \neq a_i)$ belongs $\mathcal{C}$.*

Then the communication complexity of computing $\varphi := \exists z_1, \dots, z_b \bigwedge_{C \in \mathcal{C}} C$ with respect to the partition $(X^{(1)} \setminus Z, X^{(2)} \setminus Z)$ is at least t.

Proof. Let us construct a fooling set of size 2^t. The elements of our fooling set are indexed by a binary string $r \in \{0,1\}^t$. Let us construct an assignment $\sigma_r : X \to \{0,1\}$ corresponding to r.

Let for $k \in [\ell]$, e_k denote the element of $\{0,1\}^l$ with $\ell - 1$ zeros and the only one on the kth place.

Let for $i \in [t]$, $k(i) := \min\{j : X_{s_i^1, j}\} \in X^{(1)}$.

- $\sigma_r(X_{s_i^1}) := a_i + (e_{k(i)} \cdot r_i)$ (here we add vectors in $\mathbb{F}_2^l$);
- $\sigma_r(X_{s_i^2}) := a_i + (e_{k(i)} \cdot (1 + r_i))$;
- $\sigma_r(X_{i_0}) := a_0$ and $\sigma_r(X_{j_0}) := a_0$.
- There are $n - 2t - 1$ other rows of the variables matrix X, let us choose some bijection between them and $\{0,1\}^\ell \setminus \{a_1, a_1 + e_{k(1)}, a_2, a_2 + e_{k(1)}, \dots, a_t, a_t + e_{k(1)}, a_0\}$ (the two sets have the same size since $\{a_0, a_1, \dots, a_t\}$ is a code with distance at least 3 and hence all binary strings in $\{a_1, a_1 + e_{k(1)}, a_2, a_2 + e_{k(1)}, \dots, a_t, a_t + e_{k(1)}, a_0\}$ are distinct) and σ_r substitutes values of the variables from these rows according to this bijection.

We claim that $\{\sigma_r$ restricted to $X \setminus Z \mid r \in \{0,1\}^t\}$ is a fooling set for φ. Indeed, since $\{a_0, a_1, \dots, a_t\}$ is a code with distance at least 3, for all $r \in \{0,1\}^t$, σ_r satisfies all clauses of BPHP_n^{n+1} but $(X_{i_0} \neq a_0 \vee X_{j_0} \neq a_0)$. Hence, σ_r satisfies $\bigwedge_{C \in \mathcal{C}} C$ and, then by Lemma 5, σ_r satisfies φ.

Let p and q be different strings from $\{0,1\}^t$. W.l.o.g. assume that there exists $m \in [t]$ such that $p_m = 0$ and $q_m = 1$. Let σ' be an assignment that coincides with σ_p on $X^{(1)}$ and coincides with σ_q on $X^{(2)}$. Notice that $\sigma_p(X_{s_m^1})$ and $\sigma_q(X_{s_m^1})$ differs only on $k(m)$th bit corresponding to the variable from $X^{(1)}$, hence $\sigma'(X_{s_m^1}) = \sigma_p(X_{s_m^1}) = a_m$. Analogously, $\sigma'(X_{s_m^2}) = \sigma_q(X_{s_m^1}) = a_m$. Hence σ' falsifies clause $(X_{s_m^1} \neq a_m \vee X_{s_m^2} \neq a_m)$. This clause is in $\mathcal{C}$ and all the variables of this clause are not in Z. Hence σ' falsifies φ.

So we have verified that $\{\sigma_r$ restricted to $X \setminus Z \mid r \in \{0,1\}^t\}$ is a fooling set of size 2^t. Hence, by Lemma 8, the communication complexity of φ is at least t. ◀

▶ **Lemma 22.** *Let $\mathcal{C}$ be the strict subset of the set of clauses of BPHP_n^{n+1}. Let $Z = \{z_1, \dots, z_b\} \subseteq X$ be a set of BPHP_n^{n+1} variables. Assume that there is a set $Y \subseteq [n+1]$ such that*

1. *for all $i \in Y$, X_i does not contain variables from Z;*
2. *$|Y| \geq n / \log n$;*
3. *for all $i, j \in Y$, $\mathcal{C}$ contains all clauses representing $[X_i \neq X_j]$.*

Then any OBDD representing $\varphi := \exists z_1, \dots, z_b \bigwedge_{C \in \mathcal{C}} C$ has size at least $2^{\Omega(n/\log n)}$.

Proof of Lemma 22. Let D be an OBDD computing φ. Let π be an extension of the variables order of D to X. Consider a partition of X on two parts with respect to π such that there are exactly $\lceil |Y|/2 \rceil$ rows from Y that have at least one variable from the first part. Let $i_0, j_0 \in [n+1]$ and $a_0 \in \{0,1\}^\ell$ be such that the clause $(X_{i_0} \neq a_0 \vee X_{j_0} \neq a_0)$ does not belong to $\mathcal{C}$. Let $H \subseteq \{0,1\}^l$ be a Hamming code (see Lemma 6), $|H| \geq \Omega(n/\log n)$. $H + a_0$ is also a code of distance at least 3 and $a_0 \in H + a_0$. So we can choose distinct strings $a_1, a_2, \dots, a_t \in H + a_0 \setminus \{a_0\}$ such that $t = \Omega(n/\log n)$. There are at least $\lceil |Y|/2 \rceil - 2$ rows in $Y \setminus \{i_0, j_0\}$ that have at least one variable from the first part. And also there are at least

$\lceil |Y|/2 \rceil - 2$ rows in $Y \setminus \{i_0, j_0\}$ with all variables lying in the second part. Since $\mathcal{C}$ contains all clauses representing $[X_i \neq X_j]$ for $i \neq j \in Y$, we can apply Lemma 25 and get that the communication complexity of φ with respect to the partition is at least t. Since the partition respects the variable order of D, by Lemma 9, $|D| \geq 2^t = 2^{\Omega(n/\log n)}$. ◀

▶ **Corollary 26.** *Any* OBDD *representing* $\bigwedge_{1<i<j\leq n+1}[X_i \neq X_j]$ *has size at least* $2^{\Omega(n/\log n)}$.

4.2 Proof of Lemma 23

▶ **Lemma 27.** *Let* $F(X_2, \ldots, X_{n+1}) = \exists X_1 \bigwedge_{i=2}^{n+1}[X_1 \neq X_i]$*. Then the size of any* OBDD *for* F *is at least* $2^{\Omega(n/\log n)}$.

Proof. For all $s_2, \ldots, s_{n+1} \in \{0,1\}^l$ the equality $F(s_2, \ldots, s_{n+1}) = 1$ holds if and only if there exists a binary string $s_1 \in \{0,1\}^l$ that differs from $s_2, \ldots, s_{n+1} \in \{0,1\}^l$. Such string exists if and only if there exist two equal strings among $s_2, \ldots, s_{n+1}$. Hence $F(X_2, \ldots, X_{n+1})$ is semantically equivalent to $\neg \bigwedge_{1<i<j\leq n+1}[X_i \neq X_j]$. Hence, by Corollary 26, size of any OBDD representing F is at least $2^{\Omega(n/\log n)}$. ◀

▶ **Lemma 23.** *Let* $\mathcal{C}$ *be a proper subset of clauses from* BPHP_n^{n+1}*. Let* $Z = \{z_1, \ldots, z_b\} \subseteq X$ *be a set of* BPHP_n^{n+1} *variables. Suppose that for all* $z \in Z$ *all occurrences of variables of* Z *in* BPHP_n^{n+1} *are in clauses from* $\mathcal{C}$*. Suppose also that:*

1. *There exists a row of* X *containing exactly* k *variables from* Z*.*
2. *Variables from* Z *occur in at most* $n/2$ *rows of* X*.*

Then every OBDD *that computes* $\varphi := \exists z_1, \ldots, z_b \bigwedge_{C\in\mathcal{C}} C$ *has size at least* $\min\left\{2^{\Omega(2^k/k)}, 2^{\Omega\left(2^{l-3k}/l^2\right)}\right\}$.

Proof of Lemma 23. Let Y be the set of rows of X that do not contain variables from Z. By the condition of the lemma, $|Y| \geq n/2 + 1$. We consider two cases.

Case 1. Assume that there is $T \subseteq Y$ such that $|T| = 2^k$ and the set $A_T := \{a \in \{0,1\}^l \mid \exists i, j \in T$ such that the clause $(X_i \neq a) \vee (X_j \neq a)$ belongs to $\mathcal{C}\}$ has size less than 2^{l-k}.

In this case, we show that there exists a partial substitution ρ such that $\varphi|_\rho$ is exactly the formula from Lemma 27 applied to variables of $\mathrm{BPHP}_{2^k}^{2^k+1}$. Hence, size of any OBDD computing $\varphi|_\rho$ is at least $2^{\Omega(2^k/k)}$ and, thus, any OBDD computing φ has size at least $2^{\Omega(2^k/k)}$.

Without loss of generality, assume that the row containing exactly k variables from Z is the first row of the variables matrix and $X_{1,1}, \ldots, X_{1,k} \in Z$.

Since $|A_T| < 2^{l-k}$, there exists $b \in \{0,1\}^{l-k}$ such that there are no elements in A_T with the suffix b. Let us define ρ as follows:

- For all $i \in T \cup \{1\}$, ρ assigns b to $(X_{i,k+1}, X_{i,k+2}, \ldots, X_{i,l})$.
- For $i \in [n+1] \setminus (T \cup \{1\})$, ρ assigns to X_i different elements of $\{a \in \{0,1\}^l \mid b$ is not the suffix of $a\}$.

We split $\mathcal{C}$ into three parts $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{C}_3$, where

- $\mathcal{C}_1$ consists of all clauses of $\mathcal{C}$ with all variables from the set of rows T;
- $\mathcal{C}_2$ consists of all clauses of $\mathcal{C}$ containing variables from the first row and from some row from T;
- $\mathcal{C}_3$ consists of all clauses of $\mathcal{C}$ containing variables from the set of rows $[n+1] \setminus (T \cup \{1\})$.

By the properties of φ, φ is equivalent to

$$\bigwedge_{C\in\mathcal{C}_1} C \wedge \exists X_{1,1}\dots\exists X_{1,k}\left(\bigwedge_{C\in\mathcal{C}_2} C \wedge \underset{z\in Z\setminus\{X_{1,1},\dots,X_{1,k}\}}{\exists} z \bigwedge_{C\in\mathcal{C}_3} C\right).$$

Since the support of ρ does not contain variables from $\{X_{1,1},\dots,X_{1,k}\}$, $\varphi|_\rho$ is equivalent to

$$\bigwedge_{C\in\mathcal{C}_1} C|_\rho \wedge \exists X_{1,1}\dots\exists X_{1,k}\left(\bigwedge_{C\in\mathcal{C}_2} C|_\rho \wedge \left(\underset{z\in Z\setminus\{X_{1,1},\dots,X_{1,k}\}}{\exists} z \bigwedge_{C\in\mathcal{C}_3} C\right)\Bigg|_\rho\right).$$

Since all $a\in A_T$ do not have suffix b, for all $C\in\mathcal{C}_1$, $C|_\rho = 1$.

Consider a clause $C := (X_i \neq a)\vee(X_j\neq a)\in\mathcal{C}_3$. If $i,j\in[n+1]\setminus(T\cup\{1\})$, then ρ substitute to X_i and X_j different values, hence $C|_\rho = 1$. If $i\in(T\cup\{1\})$ and $j\in[n+1]\setminus(T\cup\{1\})$, then ρ substitutes b to $(X_{i,k+1},\dots X_{i,l})$ and something different from b to $(X_{j,k+1},\dots X_{j,l})$, hence $C|_\rho = 1$. Thus, ρ satisfies $\bigwedge_{C\in\mathcal{C}_3} C$, hence by Lemma 5, ρ satisfies $\underset{z\in Z\setminus\{X_{1,1},\dots,X_{1,k}\}}{\exists} z \bigwedge_{C\in\mathcal{C}_3} C$.

So we get that $\varphi|_\rho$ is equivalent to $\exists X_{1,1}\dots\exists X_{1,k}\bigwedge_{C\in\mathcal{C}_2} C|_\rho$.

Since $\mathcal{C}$ contains all clauses of BPHP^{n+1}_n with variables from Z, $\mathcal{C}_2 = \bigwedge_{i\in T, a\in\{0,1\}^l}(X_1\neq a\vee X_i\neq a)$. If b is not a suffix of a, then ρ satisfies $(X_1\neq a\vee X_i\neq a)$. Hence, $\bigwedge_{C\in\mathcal{C}_2} C|_\rho$ is equivalent to $\bigwedge_{i\in T, a\in\{0,1\}^k}((X_{1,1},\dots,X_{1,k})\neq a\vee(X_{i,1},\dots,X_{i,k})\neq a)$. So $\varphi|_\rho$ satisfies the conditions of Lemma 27.

Case 2. Assume that for all $T\subseteq Y$ such that $|T| = 2^k$, the set $A_T := \{a\in\{0,1\}^l \mid \exists i,j\in T$ such that the clause $((X_i\neq a)\vee(X_j\neq a))$ belongs to $\mathcal{C}\}$ has size at least 2^{l-k}. Hence, for every $T\subseteq Y$ such that $|T| = 2^k$ there are $i,j\in T$ such that the set $A_{i,j} := \{a\in\{0,1\}^l \mid$ the clause $((X_i\neq a)\vee(X_j\neq a))$ belongs to $\mathcal{C}\}$ has size at least 2^{l-3k}.

In this case, we will obtain a lower bound by Lemma 25.

Let $i_0, j_0\in[n+1]$ and $a_0\in\{0,1\}^\ell$ be such that the clause $(X_{i_0}\neq a_0\vee X_{j_0}\neq a_0)$ does not belong to $\mathcal{C}$.

Let D be an OBDD computing φ. Let π be an extension of the variables order of D to X. Let $W = Y\setminus\{i_0,j_0\}$; $|W|\geq n/2-1$. Consider the following order on the set of rows $[n+1]$: we say that ith row is less than jth row if the π-minimal variable of X_i is π-less than the π-minimal variable of X_j. Let us order W according to this order: $W = \{w_1, w_2,\dots,w_{|W|}\}$. Let $d := \left\lfloor\frac{|W|}{2^k}\right\rfloor$. For every $i\in[d]$, consider the set $L_i = \{w_i, w_{i+d}, w_{i+2d},\dots,w_{i+(2^k-1)d}\}$. Since for all $i\in[d]$, $|L_i| = 2^k$ and $L_i\subseteq Y$, hence there are $e_i, f_i\in[2^k]$ such that $e_i < f_i$ and $|A_{w_{i+(e_i-1)d}, w_{i+(f_i-1)d}}|\geq 2^{l-3k}$. By the pigeonhole principle, there is $f\in[2^k]$ such that $|\{i\in[d]\mid f_i = f\}|\geq\frac{d}{2^k}$. Let us denote $I := \{i\in[d]\mid f_i = f\}$; $|I|\geq\frac{d}{2^k}\geq 2^{l-2k-1}-1$.

Let us split the set X into two parts according to π: the first part consists of all variables that are π-less than the π-minimal variable of the row $w_{1+(f-1)d}$ and the second part consists of all other elements. By the construction for all $i\in I$, the row $s^1_i := w_{i+(e_i-1)d}$ contains at least one variable from the first part (π-minimal variable of this row) and all variables from the row $s^2_i := w_{i+(f-1)d}$ are in the second part.

To apply Lemma 25, we need to show that there exist $a_i\in A_{s^1_i, s^2_i}$ such that a_0 and a_i for $i\in I$ are on the pairwise Hamming distance at least 3. We will choose them one by one; assume that we have already chosen $a_0, a_1\in A_{s^1_{i_1}, s^2_{i_1}},\dots,a_q\in A_{s^1_{i_q}, s^2_{i_q}}$ such that Hamming distance between each pair is at least 3. If $(1+l+l(l-1)/2)q < 2^{l-3k}$, then we can choose $a_{q+1}\in A_{s^1_{i_{q+1}}, s^2_{i_{q+1}}}$. So we can choose t elements if $t = \left\lfloor\frac{2^{l-3k}}{1+l+l(l-1)/2}\right\rfloor - 1$.

Hence, by Lemma 25, the communication complexity of φ with respect to the descried partition of X is at least $\Omega\left(2^{l-3k}/l^2\right)$. Then by Lemma 9, the size of D is at least $2^{\Omega\left(2^{l-3k}/l^2\right)}$. ◀

4.3 Proof of Lemma 24

▶ **Lemma 24.** *Let $\mathcal{C}$ be a strict subset of the set of clauses of* BPHP_n^{n+1}*. Let* $Z = \{z_1, \ldots, z_b\} \subseteq X$ *be a set of* BPHP_n^{n+1} *variables. Suppose that for all* $z \in Z$ *all clauses* BPHP_n^{n+1} *having variables from* Z *are in* $\mathcal{C}$*. Let* $d \in [l-1]$ *and* $T \in [2^l]$ *be such that*

1. *variables from* Z *occur in exactly* T *rows of the variables matrix* X*;*
2. *each row of* X *contains at most* d *variables from* Z*;*
3. *there exists a code* $ECC \subseteq \{0,1\}^l$ *with distance* $d+3$ *and size* $|ECC| = T+1$*;*
4. $1 + (l+1) \cdot T \cdot 2^{d+1} < n/2$.

Then every OBDD *that computes* $\varphi := \exists z_1, \ldots, z_b \bigwedge_{C \in \mathcal{C}} C$ *has size at least* 2^T*.*

Proof of lemma 24. There exist $i_0, j_0 \in [n+1]$ and a binary string $a_0 \in \{0,1\}^l$ such that $\mathcal{C}$ does not contain the clause $(X_{i_0} \neq a_0 \vee X_{j_0} \neq a_0)$.

Let $ECC \subseteq \{0,1\}^l$ be a code with distance at least $d+3$ such that $|ECC| = T+1$. W.l.o.g. assume that $a_0 \in ECC$. Let us denote $A = ECC \setminus \{a_0\}$ and $A = \{a_1, \ldots, a_T\}$.

Consider an ordered binary decision diagram D computing φ. Let π be the variables order using in D extended to all variables X. We say that a variable $x \in X$ is π-first if $x \notin Z$ and it has the minimal π-number among all such variables in its row.

As in Lemma 23 we define an order on the set of rows of the variables matrix X (or equivalently on the set $[n+1]$): ith row is less than the jth if the π-first variable of X_i is π-less the π-first variable of X_j. Let us order the elements of $[n+1] \setminus \{i_0, j_0\}$ according to this order: $\{s_1, \ldots, s_{n-1}\}$.

Now we split the variables of X into two parts such that all variables in the first part precede all variables in the second part according to the order π. The first part consists of all variables that are π-less-or-equal to the π-first variable of the row number $s_{\lfloor (n-1)/2 \rfloor}$. The second part consists of the other variables.

To prove the lower bound we will define 2^T partial substitutions ρ_α defined on the variables from the first part such that for $\alpha \neq \beta$, $\varphi|_{\rho_\alpha}$ and $\varphi|_{\rho_\beta}$ are different Boolean functions. Hence the paths in D corresponding to different ρ_α should end in different nodes. Hence, the size of D is at least 2^T.

We call a row of X as *special* if it contains at least one variable from Z. There are exactly T special rows in X. Notice that the rows i_0 and j_0 are not special, since all clauses containing variables of special rows should be in $\mathcal{C}$.

We consider two cases depending on whether all special rows lie in $\{s_1, \ldots, s_{\lfloor (n-1)/2 \rfloor}\}$ or there exists at least one *special* row in $\{s_{\lfloor (n-1)/2 \rfloor + 1}, \ldots, s_n\}$.

For convenience we denote $S_{i \ldots j} = \{s_i, \ldots, s_j\}$.

Case 1. All *special* rows are in $S_{1 \ldots \lfloor (n-1)/2 \rfloor}$. We denote the set of special rows by $\{w_1, \ldots, w_T\} \subset S_{1 \ldots \lfloor (n-1)/2 \rfloor}$.

For every $\alpha \in \{0,1\}^T$ we define a substitution ρ_α into the variables of the first part.

- Let $k(i)$ denote the index of π-first variable of the w_ith row. ρ_α substitutes to the variables of X_{w_i} from the first part corresponding values of the binary string $a_i + e_{k(i)} \cdot \alpha_i$; recall that for $k \in [\ell]$, e_k denote the element of $\{0,1\}^l$ with $\ell - 1$ zeros and the only one on the kth place. We notice that the variable $X_{w_i, k(i)}$ is necessarily in the first path.
- If $i \in \{i_0, j_0\} \cap S_{1 \ldots \lfloor (n-1)/2 \rfloor}$, then ρ_α substitutes to the variables from X_i in the first path corresponding values from the binary string a_0.
- Let $J_i := \{j \in [\ell] \mid X_{w_i, j} \in Z\}$. For every non-special row j from $S_{1 \ldots (n-1)/2}$ we choose a unique string b_j such that $b_j \neq a_0$ and for all $i \in [k]$ there exists $r \in [\ell] \setminus (J_i \cup \{k(i)\})$ such that the rth bit of b_j differs from the rth bit of a_i. Since $|J_i| \leq d$ and $T2^{d+1} + 1 < n/2$,

such strings indeed exist. We assume that the choice of b_j does not depend on α. For all $j \in S_{1\dots(n-1)/2}$, ρ_α substitutes to the variables from X_j in the first path corresponding values from b_j.

- Notice that rows from $S_{\lfloor(n-1)/2\rfloor+1\dots n-1}$ do not contain variables from the first part.

Now we show that for every $\alpha \neq \beta \in \{0,1\}^T$ functions $\varphi|_{\rho_\alpha}$ and $\varphi|_{\rho_\beta}$ are different. To do it we construct a substitution ρ to the variables of the second part such that ρ sets one of the functions to zero and the other to one. Since $\rho_\alpha \neq \rho_\beta$ then, without loss of generality, we assume that there exists row $m \in [T]$ such that $\alpha_m = 0$ and $\beta_m = 1$.

We define ρ as follows:

- For each *non-special* row $j \in S_{1\dots\lfloor(n-1)/2\rfloor}$, ρ substitutes to the second part variables from X_j values from b_j.
- For each *special* row w_i for $i \in [T]$, ρ substitutes to the second part variables from X_{w_i} values from a_i.
- For each row from $S_{\lfloor(n-1)/2\rfloor+1\dots\lfloor(n-1)/2\rfloor+2^{|J_m|}}$, ρ substitutes to the variable of the second part corresponding values of an element from $V(a_m, J_m) := \{s \in \{0,1\}^l \mid s$ agrees with a_m on $[l] \setminus J_m\}$. Since both the sets $S_{\lfloor(n-1)/2\rfloor+1\dots\lfloor(n-1)/2\rfloor+2^{|J_m|}}$ and $V(a_m, J_m)$ have $2^{|J_m|}$ elements, we can assume that for different rows ρ uses different elements of $V(a_m, J_m)$.

The definition of ρ is not finished yet but we already can show the following.

▷ Claim 28. $\varphi|_{\rho_\alpha \cup \rho} = 0$

Proof. Let τ be the restriction of $\rho_\alpha \cup \rho$ to $X \setminus Z$. If we apply τ to X_{w_m} and then substitute any values into the variables of X_{w_m} from $\mathcal{Z}$, we obtain a string from $V(b_m, J_m)$. But all such strings are substituted by $\rho_\alpha \cup \rho$ into other rows of the variables matrix. Since $\mathcal{C}$ contains all the clauses of BPHP_n^{n+1} that forbid the row X_{w_m} to be equal to any other row, there **do not** exist values for variables from $\{X_{w_m,j} \mid j \in J_m\}$ such that all clauses of $\mathcal{C}$ are satisfied and hence $\varphi|_{\rho_i \cup \rho} = \varphi|_\tau = \exists z_1, \dots, z_b \bigwedge_{C \in \mathcal{C}} C|_\tau = 0$. Note that we heavily rely on the fact that all clauses containing variables from special rows are in $\mathcal{C}$. ◁

- Now we have to define ρ on other variables i.e. variables from the rows $S_{\lfloor(n-1)/2\rfloor+2^{|J_m|}+1\dots n-1}$ such that $\varphi|_{\rho_\beta \cup \rho} = 1$. Note that:
  - All strings substituted by $\rho_\beta \cup \rho$ to rows from $S_{\lfloor(n-1)/2\rfloor+1\dots\lfloor(n-1)/2\rfloor+1+2^{|J_k|}}$ are different.
  - Since $\{a_0, a_1, \dots, a_T\}$ is a code with distance at least $d+3$ and $|J_m| \leq d$, strings from $V(a_m, J_m)$ that are substituted by $\rho_\beta \cup \rho$ to rows $S_{\lfloor(n-1)/2\rfloor+1\dots\lfloor(n-1)/2\rfloor+1+2^{|J_k|}}$ differ from the strings substituted by $\rho_j \cup \rho$ to rows $\{w_j | j \in [T] \setminus m\}$ and a_0.
  - All strings from $V(a_m, J_m)$ are different from $a_m + e_{k(m)} \cdot \beta_m = a_m + e_{k(m)}$ in the $k(m)$th bit.
  - No string from $V(a_m, J_m)$ can be substituted by $\rho_\beta \cup \rho$ to non-special rows.

So far $\rho_\beta \cup \rho$ substitutes values to variables from several rows. Notice that every binary string except a_0 is used at most once and a_0 is used twice (for rows i_0 and j_0). First of all, we can extend $\rho_j \cup \rho$ to all variables such that we will have only two equals rows (i_0 and j_0 equals a_0). Hence, $\rho_\beta \cup \rho$ satisfies $\bigwedge_{C \in \mathcal{C}} C$, thus by Lemma 5, $\rho_\beta \cup \rho$ satisfies φ.

Case 2. There exists $w_{k_0} \in S_{\lfloor (n-1)/2 \rfloor+1 \dots n-1}$ for some $k_0 \in [T]$. We fix an arbitrary T non-special rows $f_1, f_2, \dots, f_T$ from $S_{1 \dots \lfloor (n-1)/2 \rfloor}$ (it can be done since $T < n/4$).

For every $\alpha \in \{0,1\}^T$ we define a substitution ρ_α to the variables of the first part:

- Let now $k(i)$ denote the index of π-first variable of the f_ith row. For every $i \in [T]$, ρ_α substitutes to the variables of X_{f_i} from the first part corresponding values of the binary string $a_i + e_{k(i)} \cdot \alpha_i$; recall that for $k \in [\ell]$, e_k denote the element of $\{0,1\}^l$ with $\ell - 1$ zeros and the only one on the kth place. We notice that the variable $X_{f_i, k(i)}$ is necessarily in the first part.
- If $i \in \{i_0, j_0\} \cap S_{1 \dots \lfloor (n-1)/2 \rfloor}$, then ρ_α substitutes to the variables from X_i in the first part corresponding values from the binary string a_0.
- We say that a binary string $s \in \{0,1\}^l$ is *bad* if $s = a_0$ or there exist $b \in \{0,1\}^l$ with the Hamming distance at most 1 from s and $i \in [T]$ such that b agrees with a_i on the set of bits $J_{k_0} \cup \{k(i)\}$. The number of bad strings is at most $1 + (l+1) \cdot T \cdot 2^{d+1} < n/2$. So for every row $i \in S_{1 \dots \lfloor (n-1)/2 \rfloor} \setminus \{f_1, \dots, f_T, i_0, j_0\}$ we can choose not bad string b_i by some fixed way that is not dependent on α. ρ_α substitutes to the first part variables from X_i corresponding values from b_i.

For every distinct $\alpha, \beta \in \{0,1\}^T$ we build a substitution ρ into the variables of the second part that separates $\varphi|_{\rho_\alpha}$ and $\varphi|_{\rho_\beta}$. Without loss of generality assume that there exists index $m \in [T]$ such that $\alpha_m = 0$ and $\beta_m = 1$.

We choose arbitrarily $2^{|J_{k_0}|} - 1$ non-special rows $g_1, \dots, g_{2^{|J_{k_0}|}-1}$ in $S_{\lfloor (n-1)/2 \rfloor \dots n-1}$. Recall that $k(m)$ is the index of the π-first variable of the row f_m. Consider two sub-cases:

Case 2.1. $k(m) \notin J_{k_0}$. Let us define a partial substitution ξ; the substitution ρ will be an extension of ξ.

- For every $i \in [T]$, ξ substitutes to the variables of X_{f_i} from the second part corresponding values of the binary string a_i
- If $i \in \{i_0, j_0\}$, then ξ substitutes to the variables from X_i in the second part corresponding values from the binary string a_0.
- For $i \in S_{1 \dots \lfloor (n-1)/2 \rfloor} \setminus \{f_1, \dots, f_T, i_0, j_0\}$, ξ substitutes to the second part variables from X_i corresponding values from b_i.
- ξ substitutes to variables from $X_{w_{k_0}}$ corresponding values from a_m.
- To the rows $g_1, \dots, g_{2^{|J_{k_0}|}-1}$, ξ substitutes distinct values from $V(a_m, J_{k_0}) \setminus \{a_m\}$. (Notice that both sets have the same cardinality $2^{|J_{k_0}|} - 1$).

$\triangleright$ Claim 29. $\varphi|_{\rho_\alpha \cup \xi} = 0$.

Proof. The proof is similar to the proof of Claim 28. Recall that J_{k_0} are indices of Z-variables from w_{k_0}. Let τ be restriction of $\rho_\alpha \cup \xi$ to non-Z variables; τ substitutes to non-Z variables of w_{k_0} values from a_m. All strings from $V(a_m, J_{k_0})$ are already substituted by $\rho_\alpha \cup \xi$ to the rows $g_1, \dots, g_{2^{|J_{k_0}|}-1}$ and f_m. So, for every fixed values of Z-variables, $\rho_\alpha \cup \xi$ falsifies some clause from $\mathcal{C}$. Hence by the same reasons as in Claim 28, $\varphi|_\tau = \varphi|_{\rho_i \cup \xi} = 0$. $\triangleleft$

Now we have to define ρ by extending ξ on other variables to ensure that $\varphi|_{\rho_\beta \cup \rho} = 1$. Since $k(m) \notin J_{k_0}$, $\rho_\beta \cup \xi$ does not substitutes an element of $V(a_m, J_{k_0})$ to X_{f_m}, hence all strings that are substituted by $\rho_\beta \cup \xi$ so far are distinct except a_0 that is substituted two times: to rows X_{i_0} and Y_{i_0}. But $\mathcal{C}$ does not forbid to have X_{i_0} and X_{j_0} with value a_0. So we can continue ξ to ρ to satisfy ϕ as we did in Case 1.

Case 2.2. $k(m) \in J_{k_0}$. We first try to define ρ as an extension of ξ as in Case 2.1:

By Claim 29, $\varphi|_{\rho_\alpha \cup \xi} = 0$. But now we have a problem with extending ξ to ρ such that $\varphi|_{\rho_\beta \cup \rho} = 1$ since in this case $\rho_\beta \cup \xi$ substitutes to X_{f_m} the string $a_m + e_{k(m)}$ from $V(a_m, J_{k_0})$ and, hence, $\rho_\beta \cup \xi$ substitutes the same string into two different rows of the matrix: f_k and some row from $\{g_1, \dots, g_{2^{|J_{k_0}|}-1}\}$. If it is not forbidden by clauses from $\mathcal{C}$ then we can continue ξ to ρ such that $\varphi|_{\rho_j \cup \rho} = 1$ as in the previous case.

Otherwise, we redefine ξ in the following way: we flip the value of ξ on some non-Z variable from $X_{w_{k_0}}$. Now all strings that are substituted by $\rho_i \cup \xi$ are different (except a_0) since only a_m and a_0 were substituted two times by the old version of ξ. Here we use the fact that the new string substituted to w_{k_0} is bad since it is within distance 1 from a_m. Hence we can continue ξ to ρ such that $\varphi|_{\rho_\alpha \cup \rho} = 1$. But now $\varphi|_{\rho_\beta \cup \rho} = 0$ since two different non-special rows are substituted with $a_m + e_{k(m)}$ and it is forbidden by a clause from φ. ◀

5 Further Research

In this section, we would like to highlight a few open questions that naturally follow from our current research:

1. Prove a super-polynomial lower bound for the OBDD($\wedge, \exists$, reordering) proof system.
2. Is BPHP_n^{n+1} hard for the OBDD($\wedge, \exists$) proof system?
3. Is it possible to separate the OBDD($\wedge, \exists$) proof system and OBDD($\wedge, \exists$) algorithms?

References

1 Alfonso San Miguel Aguirre and Moshe Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, pages 121–136, 2001. doi:10.1007/3-540-45578-7_9.

2 Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning*, 35(1-3):51–72, 2005. doi:10.1007/s10817-005-9006-x.

3 Albert Atserias, Phokion G. Kolaitis, and Moshe Y. Vardi. Constraint propagation as a proof system. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, pages 77–91, 2004. doi:10.1007/978-3-540-30201-8_9.

4 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

5 Sam Buss, Dmitry Itsykson, Alexander Knop, Artur Riazanov, and Dmitry Sokolov. Lower bounds on OBDD proofs with several orders. *ACM Trans. Comput. Log.*, 22(4):26:1–26:30, 2021. doi:10.1145/3468855.

6 Sam Buss, Dmitry Itsykson, Alexander Knop, and Dmitry Sokolov. Reordering rule makes OBDD proof systems stronger. In Rocco A. Servedio, editor, *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 16:1–16:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CCC.2018.16.

7 Wěi Chén and Wenhui Zhang. A direct construction of polynomial-size OBDD proof of pigeon hole problem. *Information Processing Letters*, 109(10):472–477, 2009. doi:10.1016/j.ipl.2009.01.006.

8 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *The Journal of Symbolic Logic*, 44(1):36–50, 1979. URL: http://www.jstor.org/stable/2273702.

9 E. N. Gilbert. A comparison of signalling alphabets. *The Bell System Technical Journal*, 31(3):504–522, 1952. doi:10.1002/j.1538-7305.1952.tb01393.x.

10 Ludmila Glinskih and Dmitry Itsykson. Satisfiable Tseitin formulas are hard for nondeterministic read-once branching programs. In Kim G. Larsen, Hans L. Bodlaender, and Jean-François Raskin, editors, *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark*, volume 83 of *LIPIcs*, pages 26:1–26:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.MFCS.2017.26.

11 Ludmila Glinskih and Dmitry Itsykson. On tseitin formulas, read-once branching programs and treewidth. *Theory Comput. Syst.*, 65(3):613–633, 2021. doi:10.1007/S00224-020-10007-8.

12 Venkatesan Guruswami. List decoding from erasures: bounds and code constructions. *Institute of Electrical and Electronics Engineers. Transactions on Information Theory*, 49(11):2826–2833, 2003. doi:10.1109/tit.2003.815776.

13 R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. doi:10.1002/j.1538-7305.1950.tb00463.x.

14 Dmitry Itsykson, Alexander Knop, Andrei E. Romashchenko, and Dmitry Sokolov. On OBDD-based algorithms and proof systems that dynamically change the order of variables. *J. Symb. Log.*, 85(2):632–670, 2020. doi:10.1017/JSL.2019.53.

15 Jan Krajícek. An exponential lower bound for a constraint propagation proof system based on ordered binary decision diagrams. *The Journal of Symbolic Logic*, 73(1):227–237, 2008. doi:10.2178/jsl/1208358751.

16 Eyal Kushilevitz and Noam Nisan. *Communication complexity.* Cambridge University Press, 1997.

17 Chunxiao Li, Noah Fleming, Marc Vinyals, Toniann Pitassi, and Vijay Ganesh. Towards a complexity-theoretic understanding of restarts in SAT solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 2020. doi:10.1007/978-3-030-51825-7_17.

18 Stefan Mengel. Bounds on BDD-based bucket elimination. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 16:1–16:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SAT.2023.16.

19 Guoqiang Pan and Moshe Y. Vardi. Symbolic techniques in satisfiability solving. *Journal of Automated Reasoning*, 35(1-3):25–50, 2005. doi:10.1007/s10817-005-9009-7.

20 Nathan Segerlind. On the relative efficiency of resolution-like proofs and ordered binary decision diagram proofs. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity, CCC 2008, 23-26 June 2008, College Park, Maryland, USA*, pages 100–111. IEEE Computer Society, 2008. doi:10.1109/CCC.2008.34.

21 R. R. Varshamov. The evaluation of signals in codes with correction of errors. *Dokl. Akad. Nauk SSSR*, 117:739–741, 1957.

22 Ingo Wegener. *Branching Programs and Binary Decision Diagrams.* SIAM, 2000.

The Strength of the Dominance Rule

Leszek Aleksander Kołodziejczyk
Institute of Mathematics, University of Warsaw, Poland

Neil Thapen
Institute of Mathematics, Czech Academy of Sciences, Prague, Czech Republic

Abstract

It has become standard that, when a SAT solver decides that a CNF Γ is unsatisfiable, it produces a certificate of unsatisfiability in the form of a refutation of Γ in some proof system. The system typically used is DRAT, which is equivalent to extended resolution (ER) – for example, until this year DRAT refutations were required in the annual SAT competition. Recently [Bogaerts et al. 2023] introduced a new proof system, associated with the tool VeriPB, which is at least as strong as DRAT and is further able to handle certain symmetry-breaking techniques. We show that this system simulates the proof system G_1, which allows limited reasoning with QBFs and forms the first level above ER in a natural hierarchy of proof systems. This hierarchy is not known to be strict, but nevertheless this is evidence that the system of [Bogaerts et al. 2023] is plausibly strictly stronger than ER and DRAT. In the other direction, we show that symmetry-breaking for a single symmetry can be handled inside ER.

2012 ACM Subject Classification Theory of computation → Proof complexity

Keywords and phrases proof complexity, DRAT, symmetry breaking, dominance

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.20

Related Version *Full Version*: `https://arxiv.org/abs/2406.13657` [22]

Funding *Neil Thapen*: Supported by the Institute of Mathematics, Czech Academy of Sciences (RVO 67985840) and GAČR grant 23-04825S.

Acknowledgements We are grateful to Jakob Nordström for introducing us to this topic and answering our questions about it, and to Sam Buss and Vijay Ganesh for other helpful discussions.

1 Introduction

We write Lit for the set of propositional literals and $0, 1$ for the propositional constants. A *clause* is a disjunction which may contain 0, 1 or a literal together with its negation; in the last two cases we call it *tautologous*. A *conjunctive normal form* formula, or CNF, is a set of clauses, understood as a conjunction. We write $\bot$ for the empty clause and $\top$ for the empty CNF. For a clause $C = x_1 \vee \cdots \vee x_k$ we write $\neg C$ for the CNF $\neg x_1 \wedge \cdots \wedge \neg x_k$.

A *substitution* is a map $\omega : \mathrm{Lit} \cup \{0,1\} \to \mathrm{Lit} \cup \{0,1\}$ which respects negations and is the identity on $\{0,1\}$. We view a partial assignment as a kind of substitution. For literals $p_1, \ldots, p_k$ we write $\vec{p}{\restriction_\omega}$ for the tuple $\omega(p_1), \ldots, \omega(p_k)$. For a clause C, we write $C{\restriction_\omega}$ for the clause $\{\omega(p) : p \in C\}$. For a CNF Γ, we write $\Gamma{\restriction_\omega}$ for $\{C{\restriction_\omega} : C \in \Gamma\}$. We write $\omega \vDash \Gamma$ to mean that $\Gamma{\restriction_\omega}$ is tautologous, that is, every clause is tautologous. If ω is a partial assignment this is the same as the usual meaning of $\vDash$. The composition of substitutions τ, ω is defined by $\tau \circ \omega(p) = \tau(\omega(p))$. Note that $\Gamma{\restriction_{\tau\circ\omega}} = (\Gamma{\restriction_\omega}){\restriction_\tau}$ and thus $\tau \vDash \Gamma{\restriction_\omega}$ if and only if $\tau \circ \omega \vDash \Gamma$.

▶ **Definition 1.** *A* symmetry *of a CNF Γ is a substitution ω such that $\Gamma{\restriction_\omega} = \Gamma$.*

Hard combinatorial formulas often have many symmetries. A successful heuristic to make such formulas easier for SAT solvers is *symmetry-breaking*, usually in the form of introducing a *lex-leader* constraint [12]. We illustrate this in the following proposition. Suppose Γ is a CNF in variables $z_1, \ldots, z_n$ and suppose we have a formula $[x_1, \ldots, x_n \leq_{\text{lex}} y_1, \ldots, y_n]$ expressing the lexicographic order on assignments to $\vec{x}$ and $\vec{y}$. For now we suppress technical issues of how exactly we express this, and tacitly treat $[\vec{x} \leq_{\text{lex}} \vec{y}]$ as though it is a CNF.

▶ **Proposition 2.** *If ω is a symmetry of Γ, then Γ and $\Gamma \cup [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ are equisatisfiable.*

Proof. Suppose Γ is satisfiable. Let α be a lexicographically minimal assignment such that $\alpha \vDash \Gamma$. Then by symmetry $\alpha \vDash \Gamma_{\restriction\omega}$ and thus $\alpha \circ \omega \vDash \Gamma$. By minimality we have $\alpha \leq_{\text{lex}} \alpha \circ \omega$ which, working through the definitions, gives us that $\alpha \vDash [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ as required. ◀

The new formula $\Gamma \cup [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ is potentially much easier to solve than the original Γ, because the extra constraint can substantially shrink the space of partial assignments that the solver has to search through. Note that the proof still works if we simultaneously add lex-leader constraints for several different symmetries.

It is now common for SAT solvers to have a *proof-logging* component which, when it decides that a formula Γ is unsatisfiable, will generate a certificate of unsatisfiability – in other words, a refutation of Γ. For this to be useful, it must be in a well-known proof system for which trusted software exists to verify that the refutation is correct. The standard system used for this is DRAT [30]. However, it is open to what extent DRAT can (feasibly) handle reasoning that uses symmetry-breaking, especially for more than one symmetry [16].

This issue is addressed in [4] which introduces a new proof-logging system with tools to handle quite general symmetry-breaking, extending DRAT. It builds on the system VeriPB [14] and as such uses reasoning using linear inequalities, rather than clauses; furthermore it is equipped to solve optimization problems, rather than just satisfiability.

We study the proof complexity of this system (as a refutation system for CNFs). We show that it is equivalent to G_1, a system based on limited reasoning with QBFs which is one level above extended resolution (ER) in a natural hierarchy of proof systems [24]. This is in contrast to the redundancy-based systems which have been studied recently, such as DRAT, propagation redundancy and substitution redundancy, which are in their full generality all equivalent to ER [21, 17, 5]. This may represent another step in the strength of reasoning used in SAT algorithms, like from DPLL and treelike resolution, to CDCL and resolution, to the just-mentioned systems and ER [13, 26, 1]. In particular, it is unlikely that the new system in [4] can be simulated by DRAT.

The main tool used in [4] for symmetry-breaking is called the *dominance-based strengthening rule*. It is based on the following principle, which we express here in the language of CNFs and call "informal" because we are sweeping under the carpet the exact nature of the formulas Δ and $[\vec{x} <_{\text{lex}} \vec{y}]$ (intended to express strict lexicographic order).

▶ **Proposition 3** (informal). *Let Γ, Δ be formulas, where Γ is in variables $\vec{z}$ and has the property that any assignment satisfying Γ can be extended to satisfy Δ. Suppose we have a clause C and a substitution ω such that*

$$\Gamma \wedge \Delta \wedge \neg C \vDash \Gamma_{\restriction\omega} \wedge [\vec{z}_{\restriction\omega} <_{\text{lex}} \vec{z}]. \tag{1}$$

Then Γ and $\Gamma \wedge C$ are equisatisfiable.

Proof (sketch). Suppose Γ is satisfiable. Let α be a lexicographically minimal assignment such that $\alpha \vDash \Gamma$. Extend α to $\alpha \cup \beta$ satisfying $\Gamma \wedge \Delta$. If $\alpha \cup \beta \vDash C$ then we are done. Otherwise, from the entailment (1) we know $\alpha \cup \beta \vDash \Gamma_{\restriction\omega} \wedge [\vec{z}_{\restriction\omega} <_{\text{lex}} \vec{z}]$. Letting $\alpha' = (\alpha \cup \beta) \circ \omega$, we conclude that $\alpha' \vDash \Gamma$ and $\alpha' <_{\text{lex}} \alpha$, contradicting the minimality of α. ◀

The rule says roughly: if we have derived Γ, and have available a suitable Δ, ω and a proof of entailment (1), then we can derive $\Gamma \wedge C$. (In fact, the full rule in [4] is more general, since it is not restricted to the lexicographic ordering.)

Machinery exists to study the strength of rules of this kind by studying how easy it is to prove their soundness in the first-order setting of bounded arithmetic. Specifically, to carry out the proof of Proposition 3 it is sufficient to know that a nonempty polynomial-time set (in this case the set of α such that $\alpha \vDash \Gamma$) always has a lexicographically least element. This puts the system inside the theory T^1_2, which is associated with G_1 (see below for definitions).

The harder direction, lower-bounding the strength of the rule, uses similar machinery. Consider the transformation $\alpha \mapsto \alpha'$ in the proof above. We claim that, given an arbitrary polynomial time function f, we can construct an instance of the rule where this transformation is given by f. We do this by defining Δ to compute f and store the resulting values in β, and defining ω to pull these values from β back to the $\vec{z}$ variables, as α'. Observe that in the proof, it is not actually necessary for α to be lexicographically minimal; the "local minimality" property that $\alpha \leq_{\text{lex}} f(\alpha)$ is sufficient. We show that a converse holds: that using the rule, we can find a local minimum of this kind, or to say it more precisely, we can efficiently derive a contradiction from the statement that there is no local minimum.

The problem of finding such a local minimum is known as polynomial local search, or PLS [20]. It is known that if we are only interested in sentences of low quantifier complexity, such as "CNFs in this family are unsatisfiable", then every logical consequence of "every polynomial-time set has a least element", that is, of T^1_2, is already a consequence of the apparently weaker statement "every PLS problem has a solution". Using this we can show that, roughly speaking, every CNF which can be proved unsatisfiable in T^1_2 has a short refutation in the proof system in [4], where the refutation uses an instance of the rule constructed from a PLS problem as described above. It follows that the system simulates G_1.

The rest of the paper fills in the details of these arguments. In Section 2 we recall the definitions of some proof systems we will need. These are G_1 itself; cutting planes, which is the foundation for the system in [4]; and ER, where we will need to work extensively with derivations as well as refutations. In Section 3 we define our version of the system in [4], where we have removed the machinery for handling optimization problems. We call this the *dominance proof system*, and in it the dominance-based strengthening rule can be used for rather general orderings, not just lexicographic. We will work more with a restriction of it, the *linear dominance proof system*, in which it is limited to essentially the lexicographic ordering. We also define a simpler auxiliary proof system ER-PLS, which uses clauses rather than inequalities and captures the properties of the system that are important for us (illustrated in Proposition 3). It will follow from our results that, as a system for refuting CNFs, it is equivalent to the linear dominance system. In Section 4 we describe some results we need from bounded arithmetic and give a formal definition of PLS. We then show our main result,

▶ **Theorem 4.** *The linear dominance system is equivalent to G_1.*

The proof is in three parts. Section 5 contains the main technical work of the paper, showing, in Theorem 18, that ER-PLS simulates G_1, as sketched above. Section 6 shows that the linear dominance system in turn simulates ER-PLS. For the remaining direction, that G_1 simulates linear dominance, it is enough to show that linear dominance is sound, provably in T^1_2. This is in Section 7, where we also briefly discuss the difference between linear dominance and the full dominance system.

In Section 8 we study what we *can* prove in ER about fragments of these systems. We show essentially that, in Proposition 3 above, if the mysterious formula Δ is not present then we do not need to use minimality, or even PLS, in the proof. This is because without the step where we extend α to satisfy Δ, the move from α to α' does not involve any computation, but amounts to shuffling the components of α around using the substitution ω, and for simple graph-theoretical reasons we can compute the ith iteration ω^i in polynomial time without invoking any stronger principles. In Section 8.1 we use this observation to show a technical result, that a natural weakening of the linear dominance system is already simulated by ER. In Section 8.2 we use a similar construction to show[1]:

▶ **Theorem 5.** *Define the system Q_1 as* ER *plus the power to add a lex-leader constraint for a single symmetry. Then Q_1 is simulated by* ER *(and thus by* DRAT*).*

The full paper [22] contains an appendix with some additional technical material.

We finish this section by addressing the natural question: what is this hierarchy of proof systems above ER, and why should we expect it to be strict? After all, ER is already a very strong system with many tools for proving combinatorial and algebraic statements, and seems to lie well beyond current methods for proving lower bounds [28].

ER was shown in [10] to correspond to the theory PV, which models reasoning with polynomial-time concepts (see Section 4 for definitions). In [24] the systems $G_0, G_1, G_2, \ldots$ of quantified Boolean reasoning were introduced, to correspond in the same way to a hierarchy $T^0_2, T^1_2, T^2_2, \ldots$ of bounded arithmetic theories extending PV (which we can identify with T^0_2 [19]), where T^1_2 can reason with P^{NP} concepts, T^2_2 with $\mathrm{P}^{\Sigma^p_2}$, and so on. In particular, if we ignore issues of uniformity, the unsatisfiable CNFs with short refutations in G_i capture precisely the universal sentences[2] provable in T^i_2. It is a classical result that the fragments $\mathrm{I}\Sigma_0, \mathrm{I}\Sigma_1, \ldots$ of Peano arithmetic are separated by universal sentences. Specifically, the consistency statement for $\mathrm{I}\Sigma_k$ has this form and is provable in $\mathrm{I}\Sigma_{k+1}$ but not in $\mathrm{I}\Sigma_k$ (see e.g. [15, Chapter I.4(c)]). It is expected, essentially by analogy, that the analogous theories $T^0_2, T^1_2, \ldots$ are also separated at the universal level by some kind of consistency statement, although it is known that classical consistency will not work [31].

In the case we are interested in here, of PV and T^1_2, there is some evidence of separation at the $\forall\Sigma^b_1$ level (one step above universal) since it is a logical version of the question: is the TFNP class PLS different from FP? Here we at least have a relativized separation between PLS and FP [8], although this implies nothing directly about the unrelativized theories.

2 Traditional proof systems

We require that proof systems are sound and that refutations in a given system are recognizable in polynomial time. When comparing two systems P and Q we are usually interested in their behaviour when refuting CNFs, and we use the following basic definition.

▶ **Definition 6.** *We say that Q* simulates *P if there is a polynomial-time function which, given a P-refutation of a CNF Γ, outputs a Q-refutation of Γ. Q and P are* equivalent *if they simulate each other.*

Often it will make sense to discuss not only refutations of formulas, but also *derivations* of one formula from another. We will use the notation e.g. "a derivation $\Gamma \vdash \Delta$" instead of "a derivation of Δ from Γ" and will write $\pi : \Gamma \vdash \Delta$ to express that π is such a derivation.

[1] We believe this is more general than the result about single symmetries in [16], since we handle an arbitrary symmetry, not just an involution; see the discussion in [4].

[2] That is, sentences consisting of a sequence of unbounded universal quantifiers followed by a polynomial time predicate. We could also write $\forall\Pi^b_1$.

2.1 Quantified Boolean formulas and G_1

G_1 is a fragment of G, a proof system used for reasoning with quantified Boolean formulas. We give only a brief description of G – for more details see [23, Chapter 4]. We will only consider fragments of G as systems for refuting CNFs; for comparisons of G with some other systems in the context of proving quantified Boolean formulas, see e.g. [3, 9].

A *quantified Boolean formula*, or QBF, is built from propositional variables and connectives in the usual way, and also allows quantification over Boolean variables. That is, if $F(x)$ is a QBF containing a variable x, then so are $\exists x\, F(x)$ and $\forall x\, F(x)$. In this context $\exists x$ and $\forall x$ are Boolean quantifiers and these formulas semantically have the expected meanings $F(0) \vee F(1)$ and $F(0) \wedge F(1)$. We stratify QBFs into classes called Σ^q_1, $\Pi^q_1, \Sigma^q_2, \Pi^q_2$ etc. in the usual way, by counting quantifier alternations. In particular, Σ^q_1 is the closure of the class of (quantifier-free) Boolean formulas under $\vee, \wedge$ and $\exists$. The strength of proof systems working with QBFs is that they allow us to represent an "exponential-size concept" such as $\bigvee_{\vec{a}\in\{0,1\}^n} F(\vec{a})$ with a polynomial-size piece of formal notation $\exists x_1 \dots x_n F(\vec{x})$.

The proof system G is an extension of the propositional sequent calculus. In this context a *sequent* is an expression of the form

$$A_1, \dots, A_k \longrightarrow B_1, \dots, B_\ell$$

where $A_1, \dots, A_k$ and $B_1, \dots, B_\ell$ are QBFs. Such a sequent is understood semantically to mean the same as $\bigwedge_i A_i \to \bigvee_j B_j$, and we say that an assignment *satisfies* a sequent if it satisfies this formula. A derivation in G is a sequence of sequents, each of which is either an axiom of the form $A \longrightarrow A$, or follows from one or two earlier sequents by one of the rules. These rules are sound and complete and we will not list them, as the details are not so important for us (see Section 4 for our justification of this). For $k \in \mathbb{N}$ the system G_k is the restriction of G which only allows formulas from $\Sigma^q_k \cup \Pi^q_k$ to appear in derivations.

We are interested in proof systems as ways of refuting CNFs. To turn G_1 into such a system we have the following definition, where for definiteness we think of Γ as a single QBF (rather than, say, as the cedent given by its clauses) and where $\bot$ is the empty cedent.

▶ **Definition 7.** *A G_1* refutation *of a CNF Γ is a G_1 sequent calculus derivation of the sequent $\Gamma \longrightarrow \bot$.*

2.2 Pseudo-Boolean constraints and cutting planes

Following [4], we use the term *pseudo-Boolean constraint*, or *PB constraint*, for a linear inequality with integer coefficients over 0/1-valued variables. We will sometimes call a set of such constraints a *PB formula*. PB constraints generalize clauses, since a clause $C = x_1 \vee \dots \vee x_n \vee \neg y_1 \vee \dots \vee \neg y_m$ can be expressed by a constraint of the form $x_1 + \dots + x_n + (1 - y_1) + \dots + (1 - y_m) \geq 1$. We call this constraint C^* and will also write Γ^* for the PB formula obtained by taking C^* for each clause C in a CNF Γ.

If C is a PB constraint $A\vec{x} \geq b$ we write $\neg C$ for the PB constraint $A\vec{x} \leq b - 1$. Note that, although it is semantically the same, this denotes a different piece of syntax from $\neg C$ when C is a clause. Given a substitution ω, we write $C{\restriction_\omega}$ for the PB constraint obtained by simply replacing each variable x in C with $\omega(x)$, and we use a similar notation for PB formulas.

We use *cutting planes* [11], or CP, as a derivational system for deriving one PB formula G from another PB formula F. A CP derivation is a sequence of PB constraints, including every constraint from G, such that each constraint is either from F, or is a Boolean axiom of the form $x_i \geq 0$ or $x_i \leq 1$, or follows from earlier constraints by one of the rules. These are *addition* – we can derive a new constraint by summing integral multiples of two old constraints; and *rounding* – from a constraint $dA\vec{x} \geq b$, where $d > 0$ is an integral scalar and A is an integral matrix of coefficients, we can derive $A\vec{x} \geq \lceil b/d \rceil$. We use the notation $F \vdash_{\text{CP}} G$ for CP derivations. In a formal CP derivation, coefficients are written in binary.

2.3 Extended resolution as a derivational system

For CNFs Γ and Δ, a *resolution derivation* $\Gamma \vdash \Delta$ is a sequence of clauses, beginning with the clauses of Γ and containing every clause from Δ, such that each clause in the sequence is either in the initial copy of Γ or is derived from earlier clauses by the *resolution* or *weakening* rule. Here the resolution rule derives $C \vee D$ from $C \vee x$ and $D \vee \neg x$, for any variable x, and the weakening rule derives D from C for any $D \supseteq C$ (although see below for a restriction on weakening in the context of extended resolution derivations). Because we allow propositional literals 0 and 1 to occur in clauses, we need to be able to remove them, so we also allow derivation of C from $C \vee 0$ (this can be thought of as resolution with a notional axiom "1"). A *resolution refutation* of Γ is a derivation of $\Gamma \vdash \bot$.

An *extension axiom* has the form of three clauses $\neg u \vee \neg v \vee y$, $\neg y \vee u$ and $\neg y \vee v$ which together express that y is equivalent to $u \wedge v$. The intended use is that u, v are literals and y is a newly-introduced variable. For good behaviour under restrictions we also allow extension axioms of the form $\neg u \vee y$, $\neg y \vee u$ expressing that y is equivalent to a single existing literal, and of the form y or $\neg y$ expressing that y is equivalent to a constant.

▶ **Definition 8.** *For CNFs Γ and Δ, an* extended resolution (ER) *derivation $\Gamma \vdash \Delta$ is a sequence of clauses, beginning with Γ and including every clause in Δ. Each clause in the sequence either appears in the initial copy of Γ, or is derived from earlier clauses by resolution or weakening (where weakening is not allowed to introduce a variable that has not appeared earlier in the sequence*[3]*) or by the* extension rule*, which allows us to introduce an extension axiom defining a variable that has not appeared earlier in the sequence from variables that have appeared earlier.*

Such a derivation is sound in the following sense: any assignment to all variables in Γ which satisfies Γ can be extended to an assignment to all variables in Δ which satisfies Δ. The extension axioms in the derivation tell us explicitly how to extend the assignment.

We will often need to handle many extension axioms at once:

▶ **Definition 9.** *Let $\vec{x}, \vec{y}$ be disjoint tuples of variables. We say that Δ* is a set of extension axioms over *$\vec{x}; \vec{y}$ if it can be written as a sequence of extension axioms defining variables $y_1, \ldots, y_r$ in order, where each y_i is defined in terms of variables from among $\vec{x}, y_1, \ldots, y_{i-1}$.*

Equivalently, such a Δ can be thought of as describing a Boolean circuit and asserting that, on input $\vec{x}$, the values computed at the internal nodes are $\vec{y}$. We also introduce notation for writing sets of extension axioms in this way:

▶ **Definition 10.** *Suppose $\vec{x}, \vec{y}$ are tuples of variables and C is a circuit with gates of fan-in 2. We write $[\vec{y} = C(\vec{x})]$ for the set of extension axioms over $\vec{x}; \vec{y}$ expressing that the non-input nodes of the circuit have values $\vec{y}$ on inputs $\vec{x}$ (we assume $\vec{x}, \vec{y}$ have suitable arities). If the circuit has a distinguished output node we label the corresponding variable in $\vec{y}$ as y^{out}.*

[3] This restriction on weakening is probably not strictly necessary. We include it because it has the helpful consequence that each new variable comes with an explicit definition in terms of the old variables.

The condition that the extension rule must introduce new variables has some counter-intuitive consequences, and we must take extra care when we use ER as a derivational system. For example, there are ER derivations $\top \vdash x$ and $\top \vdash \neg x$, (where $\top$ is the empty CNF) but there is no derivation $\neg x \vdash x$, even though $\neg x$ extends $\top$, and no derivation $\top \vdash x \wedge \neg x$. In many ways the new variables behave like existentially quantified bound variables.

Under some reasonable conditions on how extension variables are used, we can avoid problems related to such issues. Below we formally prove two lemmas of this kind, which we will refer to as needed. We could avoid the issue by working with some other system equivalent to ER, such as extended Frege, circuit Frege [18] or even treelike G_1, but it would then be harder to show that the resulting system is simulated by linear dominance.

We use a convention that, in the context of ER derivations, when we write an expression of the form $\Gamma(\vec{x}) \vdash \Delta$ with some variables $\vec{x}$ displayed on the left, we mean that every variable in $\vec{x}$ is treated as an "old" variable in this derivation and as such is not used as an extension variable in any instance of the extension rule, even if it does not actually appear in the CNF $\Gamma(\vec{x})$.

▶ **Lemma 11.** *Let $\Gamma(\vec{x}), A(\vec{x},\vec{y}), B(\vec{x},\vec{z}), \Delta(\vec{x},\vec{w})$ by CNFs, where we assume $\vec{x}$, $\vec{y}$, $\vec{z}$, $\vec{w}$ are disjoint and no other variables appear. Suppose we have ER derivations*

$$\pi_1 : \Gamma(\vec{x}) \wedge A(\vec{x},\vec{y}) \vdash B(\vec{x},\vec{z}) \quad \textit{and} \quad \pi_2 : \Gamma(\vec{x}) \wedge B(\vec{x},\vec{z}) \vdash \Delta(\vec{x},\vec{z},\vec{w}).$$

Then we can construct an ER derivation $\Gamma(\vec{x}) \wedge A(\vec{x},\vec{y}) \vdash \Delta(\vec{x},\vec{z},\vec{w})$ in polynomial time.

Proof. We first copy π_1. Then we copy π_2, except that every extension variable in π_2 which is not in $\vec{w}$ is given a new name, to avoid clashes with variables $\vec{y}$ and other extension variables that appeared in π_1. ◀

▶ **Lemma 12.** *Given an* ER *derivation $\pi_1 : \Gamma(\vec{x}) \vdash \Delta \wedge A$, where Δ is a set of extension axioms over $\vec{x}; \vec{y}$, we can construct in polynomial time an* ER *derivation $\pi_2 : \Gamma(\vec{x}) \wedge \Delta \vdash \Delta \wedge A$.*

Proof. Let Δ' and A' be the same as Δ and A except that we have replaced every variable y_i with a new variable z_i. From $\Gamma \wedge \Delta$, we can derive $\Delta' \wedge A'$ by a copy of π_1 with Δ added to the initial clauses and with each y_i changed to z_i everywhere outside of Δ. Then we can work through $\vec{y}$ and derive, from the relevant extension axioms in Δ and Δ', using the normal rules of resolution, that $y_i \leftrightarrow z_i$; formally, this is the two clauses $\neg y_i \vee z_i$ and $\neg z_i \vee y_i$. Finally we resolve these clauses with the clauses of A' to derive A. ◀

3 The dominance rule

We define three refutational proof systems using versions of the dominance-based strengthening rule of [4]. The *dominance* proof system is intended to be the same as the system described in [4] except that we have removed the machinery for talking about optimization, that is, everything related to the objective function f. The *linear dominance* proof system restricts this by only allowing a particular kind of ordering to be used in the dominance rule; the practical work in [4] in fact only needs this weaker system. Lastly we introduce our auxiliary system ER-PLS.

In these systems, often we can only apply a rule on the condition that some other derivation $\Gamma \vdash \Delta$ or $\mathscr{C} \vdash_{\mathrm{CP}} \mathscr{D}$ exists, in ER or CP, possibly involving formulas that do not appear explicitly in the proof we are working on; or that some other polynomial-time-checkable object exists, such as a substitution ω. To ensure that correctness of a proof is checkable in polynomial time we implicitly require that, in a formal proof in a dominance-based system, each application of the rule is labelled with an example of the object in question, with the size of the labels (that is, the CP derivations, substitutions etc.) counted towards the size of the formal proof.

3.1 The dominance proof system

This is a system for refuting PB formulas. As in [4], we will call steps in a refutation *configurations*, rather than lines. A configuration is a quadruple $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z})$ where

- $\mathscr{C}$ is a set of PB constraints called *core constraints*
- $\mathscr{D}$ is a set of PB constraints called *derived constraints*
- $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ is a PB formula where $\vec{x}$ and $\vec{y}$ both have the same arity as $\vec{z}$
- $\vec{z}$ is a tuple of variables.

We put no conditions on which variables appear in $\mathscr{C}$ and $\mathscr{D}$, except that the variables $\vec{x}, \vec{y}$ in $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ should be thought of as dummy variables that are not related to the rest of the proof. In a valid proof, in every configuration the formula $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ defines a preorder and we use this with $\vec{z}$ to define a preorder $\preceq$ on assignments, writing $\alpha \preceq \beta$ if $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ is satisfied under the assignment that takes $\vec{x}$ to $\alpha(\vec{z})$ and $\vec{y}$ to $\beta(\vec{z})$.

Semantically a configuration can be thought of as asserting that $\mathscr{C}$ is satisfiable, and that if we order assignments by $\mathscr{O}_{\preceq}$ on $\vec{z}$ as described above, then for any assignment α satisfying $\mathscr{C}$, some assignment β with $\beta \preceq \alpha$ satisfies $\mathscr{C} \cup \mathscr{D}$ (see Definition 19 below).

A refutation of a PB formula F is then a sequence of configurations, beginning with $(F, \emptyset, \top, \emptyset)$, where $\top$ is the empty PB formula, and ending with a configuration in which $\mathscr{C}$ or $\mathscr{D}$ contains the contradiction $\bot$, that is, $0 \geq 1$. Each configuration is derived from the previous configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z})$ by one of the following rules:

Implicational derivation rule. Derive $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z})$, if there is a derivation[4] $\mathscr{C} \cup \mathscr{D} \vdash_{\text{CP}} C$.

(**Objective bound update rule** – this appears in [4], but we omit it from our systems as it only affects the objective function f, which we do not use.)

Redundance-based strengthening rule. Derive $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z})$ if there is a substitution ω and a derivation $\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash_{\text{CP}} (\mathscr{C} \cup \mathscr{D} \cup \{C\})\restriction_{\omega} \cup \mathscr{O}_{\preceq}(\vec{z}\restriction_{\omega}, \vec{z})$.

Deletion rule. Derive $(\mathscr{C}', \mathscr{D}', \mathscr{O}_{\preceq}, \vec{z})$ if

1. $\mathscr{D}' \subseteq \mathscr{D}$ and
2. $\mathscr{C}' = \mathscr{C}$ or $\mathscr{C}' = \mathscr{C} \setminus \{C\}$ for some constraint C derivable by the redundance rule above from $(\mathscr{C}', \emptyset, \mathscr{O}_{\preceq}, \vec{z})$[5].

Transfer rule. Derive $(\mathscr{C}', \mathscr{D}, \mathscr{O}_{\preceq}, \vec{z})$ if $\mathscr{C} \subseteq \mathscr{C}' \subseteq \mathscr{C} \cup \mathscr{D}$. In other words, we can copy constraints from $\mathscr{D}$ to $\mathscr{C}$

Dominance-based strengthening rule. We first give a slightly informal definition: derive $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z})$ if there is a substitution ω and, informally, a derivation

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash_{\text{CP}} \mathscr{C}\restriction_{\omega} \cup (\vec{z}\restriction_{\omega} \prec \vec{z})$$

[4] In [4], the derivation is allowed to use some additional inferences beyond those of CP. For simplicity we omit these, as in the presence of the redundance-based strengthening rule, even strengthening CP here to a system like extended Frege would not make any difference to the overall dominance system. In particular, our proof of the simulation of linear dominance by G_1 in Section 7 would still go through.

[5] This restriction of the deletion rule ensures that it preserves semantic validity under the intuitive meaning of configurations mentioned above. See Section 7 for an argument.

where $\vec{z}_{\restriction\omega} \prec \vec{z}$ expresses that $\vec{z}_{\restriction\omega}$ is strictly smaller than $\vec{z}$ in the ordering $\mathscr{O}_{\preceq}$. However it may be that any PB formula expressing the strict inequality $(\vec{z}_{\restriction\omega} \prec \vec{z})$ is very large. So formally the rule is: derive $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathscr{O}_{\preceq}, \vec{z})$ if there is a substitution ω and two derivations

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vdash_{\text{CP}} \mathscr{C}_{\restriction\omega} \cup \mathscr{O}_{\preceq}(\vec{z}_{\restriction\omega}, \vec{z})$$
$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \mathscr{O}_{\preceq}(\vec{z}, \vec{z}_{\restriction\omega}) \vdash_{\text{CP}} \bot.$$

Order change rule. From $(\mathscr{C}, \emptyset, \mathscr{O}_{\preceq}, \vec{z})$ derive $(\mathscr{C}, \emptyset, \mathscr{O}'_{\preceq}, \vec{z}')$ if $\mathscr{O}'_{\preceq}$ is CP-provably a preorder. That is, if there are derivations $\emptyset \vdash_{\text{CP}} \mathscr{O}_{\preceq}(\vec{u}, \vec{u})$ and $\mathscr{O}_{\preceq}(\vec{u}, \vec{v}) \cup \mathscr{O}_{\preceq}(\vec{v}, \vec{w}) \vdash_{\text{CP}} \mathscr{O}_{\preceq}(\vec{u}, \vec{w})$.

3.2 The linear dominance proof system

This restricts the dominance proof system to only use orderings $\mathscr{O}_{\preceq}$ arising from a multilinear objective function. Formally, we require that $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ is always of the form $f(\vec{x}) \leq f(\vec{y})$, where f is a multilinear function $\vec{x} \mapsto \sum_i b_i x_i$ for some constants b_i. These constants can be changed using the order change rule. We are no longer required to explicitly prove that $\mathscr{O}_{\preceq}(\vec{x}, \vec{y})$ is an ordering, as CP can always prove this for this restricted form.

The most important ordering of this form is the lexicographic ordering, which we get by setting $f(\vec{x}) \mapsto \sum_i 2^i x_i$ (for a suitable ordering of the variables in $\vec{x}$).

3.3 The system ER-PLS

This system uses the clausal version of the dominance rule sketched in Proposition 3 in the introduction. The name is intended to suggest that it has a similar connection to polynomial local search “computations” as ER has to polynomial time.

We fix a polynomial-time constructible family of CNFs defining lexicographic ordering. That is, for each k we have a CNF $[x_1, \ldots, x_k \leq_{\text{lex}} y_1, \ldots, y_k]$, which may also use some auxiliary variables $\vec{z}$, such that for all $\alpha, \beta \in \{0,1\}^k$ there is an assignment to $\vec{z}$ satisfying $[\alpha \leq_{\text{lex}} \beta]$ if and only if $\alpha \leq \beta$ lexicographically. It is not too important which CNF we use for $[\vec{x} \leq_{\text{lex}} \vec{y}]$; for a convenient one, see the full paper.

An ER-PLS refutation of a CNF Γ is formally a sequence of CNFs, beginning with Γ and ending with a CNF containing the empty clause $\bot$. At each step we apply one of the two rules below to derive the next CNF in the sequence.

ER rule. From Γ derive $\Gamma \wedge \Delta$ if there is an ER derivation $\Gamma \vdash \Delta$, in the sense of Definition 8.

Dominance rule. Let $\vec{x}$ list all variables of Γ in some order and let C be a clause in these variables. From Γ derive $\Gamma \wedge C$, provided we have

1. a set Δ of extension axioms over $\vec{x}; \vec{y}$
2. a substitution ω mapping variables $\vec{x}$ to variables among $\vec{x} \cup \vec{y}$
3. two ER derivations
   a. $\Gamma \wedge \Delta \wedge \neg C \vdash \Gamma_{\restriction\omega}$
   b. $\Gamma \wedge \Delta \wedge \neg C \wedge [\vec{x} \leq_{\text{lex}} \vec{x}_{\restriction\omega}] \vdash \bot$

with the technical condition that the auxiliary variables $\vec{z}$ used in $[\vec{x} \leq_{\text{lex}} \vec{x}_{\restriction\omega}]$ may not appear in Γ, Δ, or C.

Informally, condition 3 can be thought of as asking for a single ER derivation $\Gamma \wedge \Delta \wedge \neg C \vdash \Gamma_{\restriction\omega} \wedge [\vec{x}_{\restriction\omega} <_{\text{lex}} \vec{x}]$, as in Proposition 3. Note that we do not have any deletion rule, that is, we can only grow working set of clauses Γ, and never shrink it. This is because Γ is modelled on the *core* constraints $\mathscr{C}$ in the dominance system, which can only be deleted in very specific situations. For simplicity we do not allow deletion at all, as we will not need it.

▶ **Lemma 13.** *If* Γ' *is derived from* Γ *by a rule of* ER-PLS*, then* Γ' *and* Γ *are equisatisfiable.*

Proof. The only nontrivial case is the forward direction of the dominance rule. This is proved in the same way as Proposition 3 in the introduction, with the cosmetic change that we now have $[\vec{x} \leq_{\mathrm{lex}} \vec{x}{\restriction_\omega}]$ on the left of the entailment rather than $[\vec{x}{\restriction_\omega} <_{\mathrm{lex}} \vec{x}]$ on the right. We must also deal now with the auxiliary variables in $[\vec{x} \leq_{\mathrm{lex}} \vec{x}{\restriction_\omega}]$, but since these are not in the domain of the ordering this presents no problem. ◀

4 Bounded arithmetic

We will carry out some arguments in theories of bounded arithmetic, which we will turn into propositional proofs using variants of well-known translations. Here we give a brief overview – for more see e.g. [23, Chapters 9 and 12]. When we write that a formula with free variables is provable in a first-order theory, we mean that its universal closure is provable.

4.1 Theories

PV is the canonical theory for polynomial-time reasoning [10]. Its language contains a function symbol, called a PV *function*, for every polynomial-time algorithm on $\mathbb{N}$. Its axioms are defining equations for all PV functions, based on Cobham's characterization of polynomial-time functions. See e.g. [23, Chapter 12.1] for a precise definition (there the theory is called PV_1). Importantly, PV proves the principle of mathematical induction for any property defined by a PV *formula* – that is, by a quantifier-free formula in the language of PV. Such formulas define precisely the polynomial-time properties.

More powerful theories can be obtained by extending PV with stronger induction axioms. A formula in the language of PV is Σ^b_1 if it has the form $\exists x \leq t\, \varphi$, where t is a term not containing x and φ is a PV formula; unsurprisingly, Σ^b_1 formulas define exactly properties in NP. A formula is Σ^b_2 if it has the form $\exists x \leq t_1\, \forall y \leq t_2\, \varphi$ for φ a PV formula. The classes Π^b_1 and Π^b_2 are defined dually.

The theory T^1_2 (a more accurate name would be $T^1_2(\mathrm{PV})$) extends PV by induction axioms for all Σ^b_1 formulas. T^1_2 is the weakest theory that suffices to prove the least number principle for Σ^b_1 formulas, that is, that every nonempty NP set has a least element; actually, even the least number principle for PV formulas already implies T^1_2 over PV [6].

The theory S^1_2, intermediate in strength between PV and T^1_2, extends PV by the *length induction* axioms for Σ^b_1 formulas, that is, universal closures of statements of the form

$$\psi(0) \wedge \forall x\,(\psi(x) \to \psi(x+1)) \to \forall x\, \psi(|x|),$$

where ψ is Σ^b_1 and $|\cdot|$ stands for the *length* function that takes a number x to its length in binary notation. The theory S^2_2 is a strengthening of T^1_2 that additionally contains length induction for Σ^b_2 formulas. It should be noted that S^1_2 proves length induction also for Π^b_1 formulas, T^1_2 proves induction also for Π^b_1 formulas, and so on.

Ordered by strength, we have $\mathrm{PV} \subseteq S^1_2 \subseteq T^1_2 \subseteq S^2_2 \subseteq \dots$. There is also partial conservativity between some adjacent theories. In particular, S^1_2 is $\forall\Sigma^b_1$-conservative over PV and S^2_2 is $\forall\Sigma^b_2$-conservative over T^1_2 [7]. This means that if $\psi(x)$ is a Σ^b_1 formula and S^1_2 proves $\forall x\, \psi(x)$, then PV proves it as well; analogously for Σ^b_2 formulas, S^2_2 and T^1_2.

There is a well-known connection between propositional proof systems and arithmetic theories, linking ER to PV (and S^1_2) and G_1 to T^1_2 (and S^2_2). The following theorem, which shows that a theory proves the soundness of the corresponding proof system, can be viewed as an upper bound: it says that, for example, G_1 is in some sense no stronger than T^1_2. We use this for our main result in Section 5.

▶ **Theorem 14** ([10, 24]). *S^1_2 – and by conservativity,* PV *– proves the CNF-reflection principle for* ER*: "any CNF refutable in* ER *is unsatisfiable". Similarly, S^2_2 – and by conservativity, T^1_2 – proves the CNF-reflection principle for G_1: "any CNF refutable in G_1 is unsatisfiable".*

Proof sketch. We first consider G_1. The statement "sequent s, containing only formulas from $\Sigma^q_1 \cup \Pi^q_1$, is satisfied by assignment α to its free variables" can be naturally written as a Π^b_2 formula $\sigma(s, \alpha)$. Using this we formalize the natural proof of the soundness of G_1 as a length induction on a Π^b_2 formula, roughly as follows: given a G_1 derivation π, we show by induction down π that $\forall \alpha\, \sigma(s, \alpha)$ holds for every line s of π. If the last line has the form $\Gamma \vdash \bot$ for a CNF Γ, this means that Γ cannot be satisfiable.

The part about ER is proved by a similar argument, but with the length induction hypothesis being "every line in the ER refutation π up to the current one is satisfiable". This can be stated in a Σ^b_1 way, holds at the beginning of π if the CNF Γ being refuted is satisfiable, but no longer holds once π reaches the empty clause $\bot$. ◀

In the other direction, it is possible to translate proofs in an arithmetic theory into uniform families of propositional proofs. We will use this translation for PV and ER, and we give a slight refinement of it in Section 4.2. Such translations can also be used to give a kind of converse to Theorem 14, with a proof similar to our approach in Section 5 below:

▶ **Theorem 15** ([10, 24]). *If* PV *(equivalently S^1_2) proves the CNF-reflection principle for a propositional proof system Q, then* ER *simulates Q. Similarly, if T^1_2 (equivalently S^2_2) proves the CNF-reflection principle for Q, then G_1 simulates Q.*

Theorems 14 and 15 together give us a close association between T^1_2 and short G_1 proofs, and in fact we prove our results about G_1 indirectly using these theorems, rather than by reasoning about G_1 itself. This is largely the reason that we did not include a complete description of G_1 in Section 2.1.

Another important property of T^1_2 – and, by conservativity, of S^2_2 – is that its $\forall\Sigma^b_1$ consequences can be witnessed by *polynomial local search.* We formally define a PLS problem as a triple (t_w, θ_w, N_w), where t, N are respectively a unary and a binary PV function, θ is a binary PV formula, and the distinguished argument w written in the subscript is an instance of the problem. The formula θ defines the domain of the problem on instance w (tacitly, any element of the domain is required to be at most polynomially larger than w); t_w is an initial value that should be in the domain; and N_w is a one-place *neighbourhood* function, which attempts to map any value in the domain to a strictly smaller value in the domain. Since the domain has a least element as long as it is nonempty, N_w will sometimes fail, and a solution to the problem on instance w is either t_w, if $\neg\theta_w(t_w)$, or a value y such that $\theta_w(y)$ but either $\neg\theta_w(N_w(y))$ or $N_w(y) \geq y$.

The PLS witnessing theorem for T^1_2, originally proved in [8], says that if T^1_2 proves $\forall w\, \exists y \leq t\, \varphi$, where φ is a PV formula, then the task of finding y given w can be reduced to a PLS problem. Written in a modern form, which also includes an upper bound on the strength of the theory needed to prove correctness of the reduction, we have:

▶ **Theorem 16** ([2, Theorem 2.5]). *Assume that $T^1_2 \vdash \forall w\, \exists y \leq t\, \varphi(w, y)$, where φ is a* PV *formula. Then there is a PLS problem $Q_w = (t_w, \theta_w, N_w)$ and a* PV *function f such that the following are provable in* PV*:*

1. $\neg\theta_w(t_w) \to \varphi(w, f(w))$
2. $\theta_w(z) \wedge \neg\theta_w(N_w(z)) \to \varphi(w, f(z))$
3. $\theta_w(z) \wedge N_w(z) \geq z \to \varphi(w, f(z))$.

4.2 Propositional translations

We use a version of the translation from PV proofs to polynomial-time constructible families of ER proofs, due to Cook [10].[6] We first describe how to translate formulas. Consider a PV formula $\theta(\vec{x})$. Let $\vec{k}$ represent a choice of binary bit-lengths for the variables $\vec{x}$ (we will not be very formal about $\vec{k}$; we can think of this notation as assigning a length to every free first-order variable in the universe). Supposing $\vec{x} = x_1, \dots, x_\ell$, we will code each x_i using a k_i-tuple of new propositional variables $x_i^1, \dots, x_i^{k_i}$, which tuple we write just as $\vec{x}_i$.

By definition, θ is a quantifier-free formula built from PV functions. So we can construct, in some canonical way based on the structure of θ, in time polynomial in the bit-lengths $k_1, \dots, k_\ell$, a Boolean circuit $C(\vec{x}_1, \dots, \vec{x}_\ell)$ evaluating θ on binary inputs of these lengths. Following Definition 10, we introduce a tuple of new variables $\vec{z}$, one for each node in C, and define the propositional translation $[\![\theta(\vec{x})]\!]_{\vec{k}}$ to be the CNF $[\vec{z} = C(\vec{x}_1, \dots, \vec{x}_\ell)] \wedge z^{\text{out}}$. Unless stated otherwise, we assume that the translations of any two explicitly listed formulas have disjoint auxiliary variables $\vec{z}$. For example, in Proposition 17, we assume that the translations of $\varphi_1, \dots, \varphi_r, \theta$ all have disjoint auxiliary variables, even if some formula appears twice in this list.

We now state how we translate proofs. For more details see the full paper.

▶ **Proposition 17.** *Suppose* PV *proves a sentence* $\forall \vec{x}, \varphi_1(\vec{x}) \wedge \dots \wedge \varphi_r(\vec{x}) \to \theta(\vec{x})$, *where* $\varphi_1, \dots, \varphi_r, \theta$ *are quantifier-free. Then for any assignment* $\vec{k}$ *of bit-lengths to the variables* $\vec{x}$, *we can construct in time polynomial in* $\vec{k}$ *an* ER *derivation*

$$[\![\varphi_1(\vec{x})]\!]_{\vec{k}} \wedge \dots \wedge [\![\varphi_r(\vec{x})]\!]_{\vec{k}} \vdash [\![\theta(\vec{x})]\!]_{\vec{k}}.$$

5 ER-PLS simulates G_1

This section contains the main technical work of the paper. We begin by constructing some PV proofs. We will use propositional translations of these in our simulation.

Let $\text{Sat}(a, x)$ be a natural PV formula expressing that CNF a is satisfied by assignment x. Let $\text{Ref}(a, b)$ be a natural PV formula expressing that b is a G_1 refutation of a. We may take $\forall a, b, x\ \neg\text{Sat}(a, x) \vee \neg\text{Ref}(a, b)$ as the CNF-reflection principle for G_1, stating that any CNF refutable in G_1 is unsatisfiable. By Theorem 14 this is provable in T_2^1.

CNF-reflection is a universally-quantified PV formula, so in particular it is $\forall\Sigma_1^b$. Thus by Theorem 16, there is a PLS problem $Q_w = (t_w, \theta_w, N_w)$, where to save space we think of a, b, x as combined into a single parameter w which we write as a subscript, such that the existence of a solution to Q_w implies $\neg\text{Sat}(a, x) \vee \neg\text{Ref}(a, b)$, provably in PV. Precisely, PV proves the following three formulas, in free variables a, b, x, y (note that since the CNF-reflection principle does not contain an existential quantifier, we do not need the function f that appears in Theorem 16):

1. $\neg\theta_w(t_w) \ \to\ (\neg\text{Sat}(a, x) \vee \neg\text{Ref}(a, b))$
2. $\theta_w(y) \wedge \neg\theta_w(N_w(y)) \ \to\ (\neg\text{Sat}(a, x) \vee \neg\text{Ref}(a, b))$
3. $\theta_w(y) \wedge N_w(y) \geq y \ \to (\neg\text{Sat}(a, x) \vee \neg\text{Ref}(a, b))$.

[6] We emphasize that we are using the Cook translation, rather than the Paris-Wilkie translation of e.g. [27]. The Paris-Wilkie translation is usually used to translate first-order proofs involving an oracle symbol into families of small proofs in relatively weak propositional systems. For example, it translates (a relativized version of) T_2^1 into polylogarithmic-width resolution.

By standard properties of PLS, we may assume in order to simplify some things below that the bit-length of t_w depends only on the components a, b of w and not on the assignment x, and that N_w is hard-wired to never give output bigger than t_w.

Making some small rearrangements and introducing a new variable u for the neighbour of y, we get that PV proves

$F1.$ $\quad \mathrm{Sat}(a,x) \wedge \mathrm{Ref}(a,b) \wedge y = t_w \;\rightarrow\; \theta_w(y)$

$F2.$ $\quad \mathrm{Sat}(a,x) \wedge \mathrm{Ref}(a,b) \wedge \theta_w(y) \wedge u = N_w(y) \;\rightarrow\; \theta_w(u)$

$F3.$ $\quad \mathrm{Sat}(a,x) \wedge \mathrm{Ref}(a,b) \wedge \theta_w(y) \wedge u = N_w(y) \wedge y \leq u \;\rightarrow\; \perp.$

With these proofs in hand we can describe the simulation.

▶ **Theorem 18.** ER-PLS *simulates* G_1.

Proof. We are given a CNF A and a G_1 refutation B of A. We want to construct, in polynomial time, an ER-PLS refutation of A. We will build the refutation using propositional translations of the proofs $F1$-$F3$ above. We begin by calculating the bit-length of the variables a, b, x, y, u which we will use in the translation.

Let n be the number of variables in A and let m and ℓ be the bit-length of the strings coding A and B respectively. We may assume $n \leq m$. We will use m, ℓ, n as the respective bounds on the bit lengths of a, b, x. By our simplifying assumption on the problem Q_w we can find a bound r, polynomial in m and ℓ, on the bit-length of t_w for parameters w of the lengths we are considering. We may also use r as the bit-length for both y and u, since N_w needs at most r bits to encode its output. Thus we use these bounds m, ℓ, n, r, r as the bit-length parameter $\vec{k}$ in all our propositional translations below. As a result, all the CNFs we obtain from the translation will have size polynomial in $m + \ell$. For simplicity of notation, we will omit actually writing the subscript $\vec{k}$.

Applying Proposition 17 to $F1$, $F2$ and $F3$, we obtain, in time polynomial in $m + \ell$, the following ER derivations:

$P1:$ $\quad [\![\mathrm{Sat}(a,x)]\!] \wedge [\![\mathrm{Ref}(a,b)]\!] \wedge [\![y = t_w]\!] \;\vdash\; [\![\theta_w(y)]\!]$

$P2:$ $\quad [\![\mathrm{Sat}(a,x)]\!] \wedge [\![\mathrm{Ref}(a,b)]\!] \wedge [\![\theta_w(y)]\!] \wedge [\![u = N_w(y)]\!] \;\vdash\; [\![\theta_w(u)]\!]$

$P3:$ $\quad [\![\mathrm{Sat}(a,x)]\!] \wedge [\![\mathrm{Ref}(a,b)]\!] \wedge [\![\theta_w(y)]\!] \wedge [\![u = N_w(y)]\!] \wedge [\![y \leq u]\!] \;\vdash\; \perp.$

These formulas and derivations are in propositional variables $\vec{a}, \vec{b}, \vec{x}, \vec{y}, \vec{u}$ that arise from a, b, x, y, u in the translation, using the bit-lengths described above (plus the requisite auxiliary and extension variables).

Now let τ be the substitution (in fact a partial assignment) that first replaces the propositional variables $\vec{a}$ and $\vec{b}$ with the actual bits of A and B. Then, recalling that the CNFs $[\![\mathrm{Sat}(a,x)]\!]$ and $[\![\mathrm{Ref}(a,b)]\!]$ are defined in terms of Boolean circuits taking $\vec{a}, \vec{b}, \vec{x}$ as input, we compute the values of all nodes in these circuits that do not depend on $\vec{x}$ (for $[\![\mathrm{Ref}(a,b)]\!]$ this means all nodes) and let τ assign those values to the corresponding auxiliary variables in the CNFs.

Applying τ to the derivations above, each of $P1{\restriction_\tau}, P2{\restriction_\tau}, P3{\restriction_\tau}$ is still a valid ER refutation. However we may delete $[\![\mathrm{Ref}(a,b)]\!]{\restriction_\tau}$ from the assumptions, since by construction τ satisfies every clause in $[\![\mathrm{Ref}(a,b)]\!]$, because B is in fact a refutation of A. Furthermore we make the following claims, where all the circuits and derivations asserted to exist are constructible in polynomial time from A and B. For the notation $[\vec{z} = C(\vec{e})]$ see Definition 10.

1. Write $A(\vec{x})$ for the CNF A with the variables renamed to $x_1, \ldots, x_n$. Then there is an ER derivation $A(\vec{x}) \vdash [\![\mathrm{Sat}(a,x)]\!]{\restriction_\tau}$.

2. There is a circuit D_θ and auxiliary variables $\vec{z}_y$ and $\vec{z}_u$ such that
   a. $[\![\theta_w(y)]\!]_{\restriction\tau}$ has the form $[\vec{z}_y = D_\theta(\vec{x},\vec{y})] \wedge z_y^{\text{out}}$
   b. $[\![\theta_w(u)]\!]_{\restriction\tau}$ has the form $[\vec{z}_u = D_\theta(\vec{x},\vec{u})] \wedge z_u^{\text{out}}$.

   Abusing notation, we may write these as $[\hat{\theta}(\vec{x},\vec{y})]$ and $[\hat{\theta}(\vec{x},\vec{u})]$.
3. There is a circuit $\hat{t}$ and auxiliary variables $\vec{z}_t$ such that there is an ER derivation $[(\vec{y},\vec{z}_t) = \hat{t}(\vec{x})] \vdash [\![y = t_w]\!]_{\restriction\tau}$.
4. There is a circuit $\hat{N}$ and auxiliary variables $\vec{z}_N$ such that there is an ER derivation $[(\vec{u},\vec{z}_N) = \hat{N}(\vec{x},\vec{y})] \vdash [\![u = N_w(y)]\!]_{\restriction\tau}$.
5. There is an ER derivation $[\vec{y} \leq_{\text{lex}} \vec{u}] \vdash [\![y \leq u]\!]_{\restriction\tau}$.

Here claim 2 is true by construction, and for claim 5 note that the CNF $[\![y \leq u]\!]$ is not changed after restricting by τ. Otherwise we appeal to the well-known strength and robustness of ER and the fact that we are able to choose how to express Sat, and even $y \leq x$, in PV. We give more details of claim 1 in the full paper.

Combining the derivations provided by the claims with $P1_{\restriction\tau}, P2_{\restriction\tau}, P3_{\restriction\tau}$ and appealing to Lemma 11 we get that the following three ER derivations can be constructed in polynomial time from A and B.

$$\begin{aligned}
R1: &\quad A(\vec{x}) \wedge [(\vec{y},\vec{z}_t) = \hat{t}(\vec{x})] \ \vdash\ [\hat{\theta}(\vec{x},\vec{y})]\\
R2: &\quad A(\vec{x}) \wedge [\hat{\theta}(\vec{x},\vec{y})] \wedge [(\vec{u},\vec{z}_N) = \hat{N}(\vec{x},\vec{y})] \ \vdash\ [\hat{\theta}(\vec{x},\vec{u})]\\
R3: &\quad A(\vec{x}) \wedge [\hat{\theta}(\vec{x},\vec{y})] \wedge [(\vec{u},\vec{z}_N) = \hat{N}(\vec{x},\vec{y})] \wedge [\vec{y} \leq_{\text{lex}} \vec{u}] \ \vdash\ \bot.
\end{aligned}$$

For example, for $R2$ we first combine the derivation in claim 1, the first identity in claim 2, and the derivation in claim 4 to derive the LHS of $P2_{\restriction\tau}$ from the LHS of $R2$ (we may need to rename some variables introduced by the extension rule to avoid clashes when we combine derivations, as in the proof of Lemma 11). By Lemma 11 we can then use $P2_{\restriction\tau}$ to derive $[\![\theta_w(u)]\!]_{\restriction\tau}$, which is precisely the RHS of $R2$ by the second identity in claim 2.

The reader should note that R1–R3 still capture the same idea that we began this section with, but now in a nonuniform version: they constitute a proof in ER that if $\vec{y}$ is a solution of a PLS problem related to Q_w, where w is the instance $(A, B, \vec{x})$, then $A(\vec{x})$ is false.

We can now describe an ER-PLS refutation of $A(\vec{x})$, and thus one of A. It will use one application of the ER rule and one of the dominance rule. We begin with $A(\vec{x})$. We then introduce the clauses $[\hat{\theta}(\vec{x},\vec{y})]$ by the ER rule. This is allowed, because we can obtain them from $A(\vec{x})$ by the following ER derivation: first write down the extension axioms $[(\vec{y},\vec{z}_t) = \hat{t}(\vec{x})]$, then use $R1$.

To finish the refutation we use the dominance rule to derive the empty clause. That is, in the rule we take the new clause C to be empty. The other ingredients are as follows.

1. We set $\Gamma := A(\vec{x}) \wedge [\hat{\theta}(\vec{x},\vec{y})]$, so Γ consists of all the clauses we have so far.
2. We let $\vec{v} := \vec{y}, \vec{z}_y, \vec{x}$ list all variables that occur in Γ. Here we deliberately put $\vec{y}$ first so that it is most significant in determining the lexicographic order of assignments to $\vec{v}$.
3. We set $\Delta := [(\vec{u},\vec{z}_N) = \hat{N}(\vec{x},\vec{y})] \wedge [\vec{z}_u = D_\theta(\vec{x},\vec{u})]$. This is a set of extension axioms over $\vec{v}; \vec{u}, \vec{z}_u, \vec{z}_N$.
4. We set ω to be the substitution which maps each variable in $\vec{y}, \vec{z}_y$ to the corresponding variable in $\vec{u}, \vec{z}_u$ and is the identity everywhere else.

The substitution ω is chosen so that $\Gamma_{\restriction\omega} = A(\vec{x}) \wedge [\hat{\theta}(\vec{x},\vec{u})]$. Also Δ and ω are chosen so that the range of ω is a subset of the variables appearing in Δ, as required by the rule.

The reader should have in mind the following informal process, as sketched in the introduction. Suppose we have an assignment α to $\vec{v}$ satisfying $\Gamma = A(\vec{x}) \wedge [\hat{\theta}(\vec{x},\vec{y})]$. By R2, we can use the circuits described in Δ to extend it to an assignment $\alpha \cup \beta$ to $\vec{v}, \vec{u}, \vec{z}_u, \vec{z}_N$

satisfying $\Gamma{\restriction_\omega} = A(\vec{x}) \wedge [\hat{\theta}(\vec{x}, \vec{u})]$, and by R3 the $\vec{u}$-part of β must be strictly smaller than the $\vec{y}$-part of α. By the construction of ω, if we let $\alpha' := (\alpha \cup \beta) \circ \omega$ then α' again satisfies $A(\vec{x}) \wedge [\hat{\theta}(\vec{x}, \vec{y})]$, with the $\vec{y}$-part of α' the same as the $\vec{u}$-part of β. In this way Δ and ω work together to simulate one step in the exponential-time algorithm to solve PLS by producing smaller and smaller "feasible solutions" $\vec{y}$ such that $[\hat{\theta}(\vec{x}, \vec{y})]$. Specifically, Δ computes the next solution and writes it on its new variables $\vec{u}$, then ω copies the values of $\vec{u}$ back to the old variables, overwriting $\vec{y}$.

Formally, to complete the proof we need to construct two ER derivations

(a) $\Gamma \wedge \Delta \vdash \Gamma{\restriction_\omega}$

(b) $\Gamma \wedge \Delta \wedge [\vec{v} \leq_{\text{lex}} \vec{v}{\restriction_\omega}] \vdash \bot$.

Strictly speaking we should also include $\neg C$ in both sets of assumptions, but since C is the empty clause omitting this makes no difference.

Writing out (a) in more detail, what we need to show is

$$A(\vec{x}) \wedge [\hat{\theta}(\vec{x}, \vec{y})] \wedge [(\vec{u}, \vec{z}_N){=}\hat{N}(\vec{x}, \vec{y})] \wedge [\vec{z}_u{=}D_\theta(\vec{x}, \vec{u})] \ \vdash \ A(\vec{x}) \wedge [\hat{\theta}(\vec{x}, \vec{u})].$$

If the clauses $[\vec{z}_u = D_\theta(\vec{x}, \vec{u})]$ were not present on the left then $R2$ would already be a derivation of this. However these clauses are part of $[\hat{\theta}(\vec{x}, \vec{u})]$, so we can use $R2$ with an appeal to Lemma 12.

For (b), we observe that $[\vec{v} \leq_{\text{lex}} \vec{v}{\restriction_\omega}]$ means precisely $[(\vec{y}, \vec{z}_y, \vec{x}) \leq_{\text{lex}} (\vec{u}, \vec{z}_u, \vec{x})]$, from which formula we can easily derive in ER that $[\vec{y} \leq_{\text{lex}} \vec{u}]$. Hence we can use $R3$ and Lemma 11. ◀

6 Linear dominance simulates ER-PLS

Recall from Section 2.2 the notation C^* and Γ^* for converting clauses and CNFs into equivalent PB constraints and formulas. We will show that, given a derivation of Δ from Γ in ER-PLS, we can construct in polynomial time a derivation of $(\Delta^*, \emptyset, \top, \emptyset)$ from $(\Gamma^*, \emptyset, \top, \emptyset)$ in the linear dominance system, which implies the simulation. So we must show how to handle the two rules of ER-PLS: the ER rule and the dominance rule.

ER rule. This follows straightforwardly by the well-known simulation of resolution by cutting planes [11] and, for extension steps, using the redundance-based strengthening rule (of the linear dominance system) and standard arguments about how to add extension axioms as redundant clauses, see e.g. [25]. A detailed proof is included in the full paper.

Dominance rule. Suppose we have a CNF Γ and a clause C, both in variables $x_1, \ldots, x_n$, plus a set Δ of extension axioms over $\vec{x}; \vec{y}$, a substitution ω mapping variables $\vec{x}$ to variables $\vec{x} \cup \vec{y}$, and two ER derivations

(a) $\Gamma \wedge \Delta \wedge \neg C \vdash \Gamma{\restriction_\omega}$

(b) $\Gamma \wedge \Delta \wedge \neg C \wedge [\vec{x} \leq_{\text{lex}} \vec{x}{\restriction_\omega}] \vdash \bot$.

We will describe a derivation from $(\Gamma^*, \emptyset, \top, \emptyset)$ of $(\Gamma^* \cup \{C^*\}, \emptyset, \top, \emptyset)$.

We can combine the derivations for (a) and (b) above to construct an ER derivation

$$\pi : \Gamma \wedge \Delta \wedge \neg C \vdash \Gamma{\restriction_\omega} \wedge [\vec{x}{\restriction_\omega} <_{\text{lex}} \vec{x}].$$

where $[\vec{x} <_{\text{lex}} \vec{y}]$ represents strict lexicographic ordering – see the full paper for details. Furthermore we may assume that π is actually a *resolution* derivation, that is, that it does not include any applications of the extension rule. This is because we can move all extension axioms introduced by that rule from the body of the derivation to Δ, preserving the order in which they appeared in the derivation. That process turns Δ into a set of extension axioms over $\vec{x}; \vec{y}, \vec{z}$, where $\vec{z}$ now includes all extension variables that were introduced in the original π, and in particular all auxiliary variables in $[\vec{x}{\restriction_\omega} <_{\text{lex}} \vec{x}]$.

To build the derivation in the linear dominance system, we first change the ordering from the trivial order $\top$ to the lexicographic order on $x_1, \dots, x_n$, with the most significant bits first. This can be done using the order change rule; see Section 3.2. So we are now in the configuration $(\Gamma^*, \emptyset, \mathcal{O}_\preceq, \vec{x})$, where $\mathcal{O}_\preceq$ is the lexicographic order.

We then derive $(\Gamma^*, \Delta^*, \mathcal{O}_\preceq, \vec{x})$, where we add each extension axiom in Δ in turn using the redundance-based strengthening rule, in the same way that we handle extension axioms in the the ER rule. We must check that we satisfy the order condition for this rule, but this is easy, since the substitutions used do not affect $\vec{x}$ variables, which are the only variables relevant to the ordering. Again the details can be found in the full paper.

Now we use dominance-based strengthening to derive $(\Gamma^*, \Delta^* \cup \{C^*\}, \mathcal{O}_\preceq, \vec{x})$. We apply the normal translation from resolution into CP to π to get

$$\Gamma^* \cup \Delta^* \cup \{(\neg C)^*\} \vdash_{\text{CP}} (\Gamma{\restriction_\omega})^* \cup [\vec{x}{\restriction_\omega} <_{\text{lex}} \vec{x}]^*.$$

It is easy to construct a short derivation $\neg(C^*) \vdash_{\text{CP}} (\neg C)^*$. We can also construct a derivation $[\vec{x}{\restriction_\omega} <_{\text{lex}} \vec{x}]^* \vdash_{\text{CP}} \mathcal{O}_\prec(\vec{x}{\restriction_\omega}, \vec{x})$ in polynomial time, where $\mathcal{O}_\prec$ is strict lexicographic ordering written in the natural way using the same multilinear function f as $\mathcal{O}_\preceq$ – see the full paper. Moreover, $(\Gamma{\restriction_\omega})^*$ is the same as $(\Gamma^*){\restriction_\omega}$. Thus we have

$$\Gamma^* \cup \Delta^* \cup \{\neg(C^*)\} \vdash_{\text{CP}} (\Gamma^*){\restriction_\omega} \cup \mathcal{O}_\prec(\vec{x}{\restriction_\omega}, \vec{x}). \tag{2}$$

Finally, from (2) we can trivially construct a derivation

$$\Gamma^* \cup \Delta^* \cup \{\neg(C^*)\} \cup \mathcal{O}_\preceq(\vec{x}, \vec{x}{\restriction_\omega}) \vdash_{\text{CP}} \bot. \tag{3}$$

The derivations (2) and (3) are what we need to apply the dominance-based strengthening rule to derive C^* (after weakening $\mathcal{O}_\prec$ in (2) to $\mathcal{O}_\preceq$). This completes the proof.

7 G_1 simulates linear dominance

For this result we will use Theorem 15, which states that for a propositional proof system Q, if S^2_2 proves the CNF-reflection principle for Q, then G_1 simulates Q. We take Q to be the linear dominance system, considered as a system for refuting CNFs. That is, a Q-refutation of a CNF Γ is a linear dominance refutation of Γ^*. Thus for the simulation it is enough to prove in S^2_2 that the existence of such a refutation of Γ^* implies that Γ is unsatisfiable.

We do this by formalizing in S^2_2 as much as we can of the proof of soundness of the dominance system from [4]. We run into a problem when dealing with the dominance rule. To show it is sound, it is enough to show that if a CNF is satisfiable, then it has a least satisfying assignment with respect to the ordering $\mathcal{O}_\preceq$. However as far as we know the general statement of this form, that an arbitrary ordering has a least element, is not provable in S^2_2, and is known to be unprovable if the ordering given is by an oracle [29]. It *is* provable in T^2_2, and hence in S^3_2, by a simple inductive argument. By the methods in this section it follows from this that the full dominance system is simulated by G_2.

To stay inside the strength of S^2_2 we chose to work with the linear dominance system instead since T^1_2, and hence also S^2_2, can prove that any nonempty set of strings has a least element in the lexicographic ordering, which is enough to prove the soundness of the dominance rule restricted to such an ordering.[7]

We use a definition from [4].

[7] Whether the dominance system is strictly stronger than linear dominance is unclear. Conceivably to bound the strength of the full dominance rule we could make use of the fact that the ordering is not given by an arbitrary relation, but by a relation which is provably an ordering in CP.

▶ **Definition 19.** *A configuration* $(\mathscr{C}, \mathscr{D}, \mathcal{O}_{\preceq}, \vec{z})$ *is called* valid *if*

1. $\mathscr{C}$ *is satisfiable*
2. *For every total assignment* $\alpha \vDash \mathscr{C}$*, there is a total assignment* β *with* $\beta \preceq \alpha$ *and* $\beta \vDash \mathscr{C} \cup \mathscr{D}$*.*

Notice that validity is a Π^b_2 condition, and in particular S^2_2 is strong enough to do all the basic reasoning we need about sums and inequalities. Working in S^2_2, suppose for a contradiction we are given a satisfiable CNF Γ and a linear dominance refutation π of Γ^*. We will use Π^b_2 length induction, taking as our inductive hypothesis that the ith configuration in π is valid. The base case is the initial configuration $(\Gamma^*, \emptyset, \top, \emptyset)$, which is valid by assumption (since any assignment satisfying Γ already satisfies Γ^*). On the other hand the final configuration in π is not valid, since in that configuration $\mathscr{C} \cup \mathscr{D}$ is not satisfiable. Therefore to derive a contradiction it is enough to show that every rule preserves validity.

Since the soundness of CP is trivially provable in PV, this is easy for the implicational derivation, transfer and order change rules. For the remaining rules, namely redundance-based strengthening, deletion and dominance-based strengthening, suppose we are at a valid configuration $(\mathscr{C}, \mathscr{D}, \mathcal{O}_{\preceq}, \vec{z})$.

Redundance-based strengthening rule. We have a substitution ω and we know

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vDash (\mathscr{C} \cup \mathscr{D} \cup \{C\})_{\restriction\omega} \cup \mathcal{O}_{\preceq}(\vec{z}_{\restriction\omega}, \vec{z}).$$

Let $\alpha \vDash \mathscr{C}$. By the inductive hypothesis there is $\beta \preceq \alpha$ such that $\beta \vDash \mathscr{C} \cup \mathscr{D}$, and we want to find $\beta' \preceq \alpha$ such that $\beta' \vDash \mathscr{C} \cup \mathscr{D} \cup \{C\}$. If $\beta \vDash C$ then we set $\beta' = \beta$. Otherwise we set $\beta' = \beta \circ \omega$, and the properties of β' follow from the assumption.

Deletion rule. The interesting case is that we derive $(\mathscr{C}', \mathscr{D}', \mathcal{O}_{\preceq}, \vec{z})$ with $\mathscr{D}' \subseteq \mathscr{D}$ and $\mathscr{C}' = \mathscr{C} \setminus \{C\}$ for some C derivable by the redundance rule from $(\mathscr{C}', \emptyset, \mathcal{O}_{\preceq}, \vec{z})$. Let $\alpha \vDash \mathscr{C}'$. If $\alpha \vDash C$ then there is nothing to show. Otherwise, using the notation of the redundance rule, we let $\alpha' = \alpha \circ \omega$ and know that $\alpha' \preceq \alpha$ and $\alpha' \vDash \mathscr{C}$. The inductive hypothesis then gives us $\beta' \preceq \alpha'$ with $\beta' \vDash \mathscr{C} \cup \mathscr{D}$, so in particular $\beta' \preceq \alpha$ and $\beta' \vDash \mathscr{C}' \cup \mathscr{D}'$.

For all rules so far, the proof that validity is preserved goes through even in PV. For the last rule we will need to minimize the value of a PV function on a polynomial-time computable set, which can be done in S^2_2, since it extends T^1_2 (see Section 4.1).

Dominance-based strengthening rule. We derive $(\mathscr{C}, \mathscr{D} \cup \{C\}, \mathcal{O}_{\preceq}, \vec{z})$, and for a given substitution ω we know

$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \vDash \mathscr{C}_{\restriction\omega} \cup f(\vec{z}_{\restriction\omega}) \leq f(\vec{z})$$
$$\mathscr{C} \cup \mathscr{D} \cup \{\neg C\} \cup \{f(\vec{z}) \leq f(\vec{z}_{\restriction\omega})\} \vDash \bot$$

where f is the linear function defining $\mathcal{O}_{\preceq}$. Let $\alpha \vDash \mathscr{C}$. We want to find $\beta' \preceq \alpha$ such that $\beta' \vDash \mathscr{C} \cup \mathscr{D} \cup \{C\}$. Let S be the set of total assignments $\preceq$-below α satisfying $\mathscr{C}$. Let β be a member of S for which $f(\beta)$ is minimal (where $f(\beta)$ stands for f applied to the $\vec{z}$ variables of β). Using the least number principle for Σ^b_1 formulas available in S^2_2, we can find such a β.

By the inductive hypothesis (that is, the validity of $(\mathscr{C}, \mathscr{D}, \mathcal{O}_{\preceq}, \vec{z})$) we may assume that $\beta \vDash \mathscr{C} \cup \mathscr{D}$. If $\beta \vDash C$ then we set $\beta' = \beta$. Otherwise let $\beta' = \beta \circ \omega$. By the first entailment in the rule, $\beta' \vDash \mathscr{C}$ and $f(\beta') \leq f(\beta)$, so $\beta' \in S$. Therefore by the minimality of $f(\beta)$ we have $f(\beta) \leq f(\beta')$. But this contradicts the second entailment.

This completes the proof that S^2_2 proves the soundness of the linear dominance system, which is thus simulated by G_1.

8 Simulations of fragments by ER

8.1 Weak linear dominance

Consider the version of the linear dominance system in which we limit the dominance-based strengthening rule by only allowing it to be applied when the set $\mathscr{D}$ of derived clauses is empty. That is, we replace it with the rule: from $(\mathscr{C}, \emptyset, \mathscr{O}_\preceq, \vec{z})$ derive $(\mathscr{C} \cup \{C\}, \emptyset, \mathscr{O}_\preceq, \vec{z})$ if there is a substitution ω and derivations

$$\mathscr{C} \cup \{\neg C\} \vdash_{\mathrm{CP}} \mathscr{C}_{\restriction\omega} \cup f(\vec{z}_{\restriction\omega}) \leq f(\vec{z})$$
$$\mathscr{C} \cup \{\neg C\} \cup \{f(\vec{z}) \leq f(\vec{z}_{\restriction\omega})\} \vdash_{\mathrm{CP}} \bot.$$

where f is the linear function defining $\mathscr{O}_\preceq$. We shall refer to this system as the *weak linear dominance system.*

▶ **Proposition 20.** *The weak linear dominance system is simulated by ER.*

To prove the simulation, we use a lemma saying that PV knows that there is a polynomial time function that lets us iterate a substitution m times, when m is given in *binary.*

▶ **Lemma 21.** *There is a polynomial time function $g(\omega, m)$ which takes as input a substitution ω on variables $z_1, \dots, z_n$ and a number m (coded in binary) and outputs the substitution ω^m. Furthermore this works provably in* PV*, that is,* $\mathrm{PV} \vdash g(\omega, m+1) = g(\omega, m) \circ \omega$.

Proof. Fix a variable z_i and consider the sequence $z_i, \omega(z_i), \dots, \omega^{2n+2}(z_i)$ as a walk through the space $\mathrm{Lit} \cup \{0,1\}$. The sequence can be produced by a PV function on input $\langle \omega, i \rangle$. By the pigeonhole principle, which is available in PV here since n is small (polynomial in the length of the input), this sequence must touch some point twice. That is, it consists of a walk of length k_i to some $u \in \mathrm{Lit} \cup \{0,1\}$, followed by a loop of some size ℓ_i, where $0 \leq k_i \leq 2n+1$ and $1 \leq \ell_i \leq 2n$. Again, the numbers k_i, ℓ_i can be computed by a PV function on input $\langle \omega, i \rangle$

Thus to compute $\omega^m(z_i)$ for $m > 2n+2$ it is enough to calculate the remainder of $m-k$ divided by ℓ: namely, $\omega^m(z_i) = \omega^{k_i + ((m-k_i) \bmod \ell_i)}(z_i)$. ◀

Proof of Proposition 20. By Theorem 15, it is enough to show that the soundness of the weak linear dominance system is provable in S^1_2. So, working in S^1_2, suppose that a CNF Γ is satisfiable but that Γ^* has a refutation π in the system. We will derive a contradiction. We will use length induction, but with a weaker inductive hypothesis than was used in Section 7 for the soundness of full linear dominance. Namely, we will show that for each configuration $(\mathscr{C}, \mathscr{D}, \mathscr{O}_\preceq, \vec{z})$ in turn in π, $\mathscr{C} \cup \mathscr{D}$ is satisfiable. Satisfiability is a Σ^b_1 property, so this is a form of length induction we can carry out in S^1_2. It yields a contradiction when we get to the last configuration in π.

The first configuration is satisfiable, by the assumption on Γ. It is easy to see that every rule, other than dominance-based strengthening, preserves satisfiability; in the case of the redundance-based strengthening rule, this is by the standard argument about composing the current assignment once with ω, if necessary.

So suppose we are dealing with the weak dominance-based strengthening rule. We have an assignment α which satisfies the current configuration $(\mathscr{C}, \emptyset, \mathscr{O}_\preceq, \vec{z})$, and we want to satisfy $(\mathscr{C} \cup \{C\}, \emptyset, \mathscr{O}_\preceq, \vec{z})$. We have a substitution ω and derivations

$$\mathscr{C} \cup \{\neg C\} \vdash_{\mathrm{CP}} \mathscr{C}_{\restriction\omega} \cup f(\vec{z}_{\restriction\omega}) \leq f(\vec{z})$$
$$\mathscr{C} \cup \{\neg C\} \cup \{f(\vec{z}) \leq f(\vec{z}_{\restriction\omega})\} \vdash_{\mathrm{CP}} \bot.$$

for a linear f. Suppose for a contradiction that $\mathscr{C} \cup \{\neg C\}$ is unsatisfiable. Then, since CP derivations are provably sound (even in PV) we know that for any assignment β, if $\beta \vDash \mathscr{C}$, then $\beta \circ \omega \vDash \mathscr{C}$ and $f(\beta \circ \omega) < f(\beta)$.

We may assume without loss of generality that f only takes values between 0 and some upper bound m. Writing α_i for $\alpha \circ \omega^i$, we use induction (rather than length induction) on i to show that for all i we have

$$\alpha_i \vDash \mathscr{C} \quad \text{and} \quad f(\alpha_i) \leq m - i.$$

By Lemma 21, this is a PV formula, so this induction can be carried out in S^1_2 (if the formula were Σ^b_1, we would only be able to use length induction). The base case $i = 0$ is true by the assumptions about α and f, and the inductive step follows from the discussion in the previous paragraphs. We conclude that $f(\alpha_{m+1}) \leq -1$, which is impossible. ◀

8.2 Symmetry breaking in ER

Let us define a proof system Q, which we could call ER plus *static symmetry breaking*. A refutation of a CNF Γ in Q consists of an initial step, in which we list a sequence of symmetries $\omega_1, \ldots, \omega_k$ of Γ and write down the corresponding lex-leader constraints (where for each constraint we use fresh auxiliary variables). This is followed by an ER refutation of Γ augmented by these constraints, that is, of $\Gamma' := \Gamma \wedge \bigwedge_i [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction \omega_i}]$.

For $k \in \mathbb{N}$ we define Q_k to be Q limited to only adding axioms for k symmetries.

▶ **Proposition 22.** *The full system Q is sound, and is simulated by G_1.*

Proof. We repeat the proof of Proposition 2, except this time we fill in some details. To prove soundness, it is enough to show that, supposing Γ is satisfiable, Γ' is satisfiable as well. Let α be a lexicographically minimal assignment to the $\vec{z}$-variables satisfying Γ. We claim that an extension of α satisfies Γ'. To see this, let ω_i be any symmetry from our list. Then $\alpha \vDash \Gamma$ implies $\alpha \vDash \Gamma_{\restriction \omega_i}$, and thus $\alpha \circ \omega_i \vDash \Gamma$. By minimality of α we have $\alpha \leq_{\text{lex}} \alpha \circ \omega_i$, and thus, extending α to β which satisfies the extension axioms in the definition of $\leq_{\text{lex}}$, we have that β satisfies the symmetry-breaking axiom $[\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction \omega_i}]$. In this way we can simultaneously satisfy such axioms for all i, by the assumption that auxiliary variables are disjoint.

For the simulation by G_1, it is enough to observe that this argument can be cast as a proof of the CNF-reflection principle for Q and carried out in T^1_2. Then we can appeal to Theorem 15. ◀

The converse direction is presumably false:

▶ **Proposition 23.** *G_1 is not simulated by Q, assuming G_1 is not simulated by* ER.

Proof. Let Γ_n be a family of CNFs which have polynomial-sized refutations in G_1 but require superpolynomial size in ER. Then it is easy to construct a polynomial-sized CNF A_n such that $\Gamma_n \cup A_n$ has no symmetries; assuming Γ_n has variables $x_1, \ldots, x_m$, a convenient example consists of clauses $x_i \vee y_1 \vee \cdots \vee y_i$ for each i, where $y_1, \ldots, y_m$ are new variables. Then G_1 refutations of Γ_n still work for $\Gamma_n \cup A_n$ (we may need to add one more weakening step). On the other hand, if π is any Q refutation of $\Gamma_n \cup A_n$, then it must be just an ER refutation, and we can turn it into an ER refutation of Γ_n by applying the restriction which sets every y_i variable to 1. Thus π must have superpolynomial size. ◀

We can now prove Theorem 5 from the introduction, that Q_1 is simulated by ER.

Proof of Theorem 5. We will show that the soundness of Q_1 is provable in S^1_2. The result then follows by Theorem 15. Let Γ be a CNF and let ω be a symmetry of Γ. Let $\Gamma' := \Gamma \wedge [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ and suppose we are given an ER refutation of Γ'. We will show, with a proof formalizable in S^1_2, that if Γ is satisfiable then so is Γ'. We can then derive a contradiction, since S^1_2 proves the soundness of ER (see Theorem 14).

Working in S^1_2, suppose $\alpha \vDash \Gamma$. As in the proof of Proposition 20 we write α_i for $\alpha \circ \omega^i$, and use the fact that by Lemma 21 this can be computed in polynomial time. Suppose for a contradiction that $\Gamma \wedge [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ is unsatisfiable. It follows that for any assignment β, if $\beta \vDash \Gamma$ then $\beta \circ \omega <_{\text{lex}} \beta$. On the other hand, if $\beta \vDash \Gamma$ then we already know $\beta \circ \omega \vDash \Gamma$, since $\Gamma_{\restriction\omega} = \Gamma$. Assuming that there are n many z-variables we have $\alpha \leq_{\text{lex}} 2^n - 1$, where we identify $2^n - 1$ with a string of 1s of length n. Thus we can reach a contradiction by a similar induction as in the proof of Proposition 20, showing inductively that for each i we have $\alpha_i \vDash \Gamma$ and $\alpha_i \leq_{\text{lex}} 2^n - i$. ◀

This proof breaks down immediately even for Q_2, since we do not have any equivalent of Lemma 21 for arbitrary compositions of two substitutions.

We briefly discuss how one could directly construct an ER refutation from a Q_1 refutation, without going through bounded arithmetic and Theorem 5. The main task is to construct a circuit C which, when given an assignment α such that $\alpha \vDash \Gamma$, outputs an assignment β such that $\beta \vDash \Gamma \wedge [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$. Furthermore this property of C must be provable in ER, in the sense that we have an ER derivation $\Gamma(\vec{x}) \wedge [\vec{z} = C(\vec{x})] \vdash \Gamma(\vec{z}) \wedge [\vec{z} \leq_{\text{lex}} \vec{z}_{\restriction\omega}]$ (where we are suppressing auxiliary variables in C and $\leq_{\text{lex}}$). We will just describe C.

We use a subcircuit which takes input $\vec{z}, i$ and computes $\alpha_i := \vec{z}_{\restriction\omega^i}$ using the algorithm for g in Lemma 21. The circuit C finds i such that the two conditions $\alpha_i \vDash \Gamma$ and $\alpha_i \leq_{\text{lex}} 2^n - i$ hold for i, but one of them fails for $i+1$, and outputs α_i. Such an i can be found by binary search, since both conditions hold for $i = 0$ and the second one must fail for $i = 2^n + 1$. Since $\alpha_i \vDash \Gamma$ and $\alpha_{i+1} = \alpha_i \circ \omega$, we have that $\alpha_{i+1} \vDash \Gamma$ as ω is a symmetry. We conclude that the second condition fails and $\alpha_{i+1} >_{\text{lex}} 2^n - i - 1$. Thus $\alpha_{i+1} \geq_{\text{lex}} \alpha_i$, meaning that $\alpha_i \leq_{\text{lex}} \alpha_i \circ \omega$ as required.

References

1 P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004. doi:10.1613/jair.1410.

2 A. Beckmann and S. R. Buss. Polynomial local search in the polynomial hierarchy and witnessing in fragments of bounded arithmetic. *Journal of Mathematical Logic*, 9(1):103–138, 2009. doi:10.1142/S0219061309000847.

3 O. Beyersdorff and J. Pich. Understanding Gentzen and Frege systems for QBF. In *Proceedings of the Annual Symposium on Logic in Computer Science (LICS '16)*, pages 146–155, 2016. doi:10.1145/2933575.2933597.

4 B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *Journal of Artificial Intelligence Research*, 77:1539–1589, 2023. doi:10.1613/jair.1.14296.

5 S. Buss and N. Thapen. DRAT and propagation redundancy proofs without new variables. *Logical Methods in Computer Science*, 17, 2021. URL: https://lmcs.episciences.org/7400.

6 S. R. Buss. *Bounded Arithmetic.* Bibliopolis, 1986.

7 S. R. Buss. Axiomatizations and conservation results for fragments of bounded arithmetic. In *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 57–84. ACM, 1990.

8 S. R. Buss and J. Krajíček. An application of Boolean complexity to separation problems in bounded arithmetic. *Proceedings of the London Mathematical Society*, 69:1–21, 1994. doi:10.1112/plms/s3-69.1.1.

9 L. Chew and M. J. H. Heule. Relating existing powerful proof systems for QBF. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '22)*, pages 10:1–10:22, 2022. doi:10.4230/LIPIcs.SAT.2022.10.

10 S. A. Cook. Feasibly constructive proofs and the propositional calculus (preliminary version). In *Annual ACM Symposium on Theory of Computing (STOC '75)*, pages 83–97. ACM, 1975. doi:10.1145/800116.803756.

11 W. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987. doi:10.1016/0166-218X(87)90039-4.

12 J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning (KR '96)*, pages 148–159, 1996.

13 M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962. doi:10.1145/368273.368557.

14 J. Elffers, S. Gocht, C. McCreesh, and J. Nordström. Justifying all differences using pseudo-Boolean reasoning. In *AAAI Conference on Artificial Intelligence*, volume 34(02), pages 1486–1494, 2020. doi:10.1609/aaai.v34i02.5507.

15 P. Hájek and P. Pudlák. *Metamathematics of First-Order Arithmetic*. Springer-Verlag, 1993. doi:10.1007/978-3-662-22156-3.

16 M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler. Expressing symmetry breaking in DRAT proofs. In *International Conference on Automated Deduction*, pages 591–606. Springer, 2015. doi:10.1007/978-3-319-21401-6_40.

17 M. J. H. Heule, B. Kiesl, and A. Biere. Strong extension-free proof systems. *Journal of Automated Reasoning*, 64(3):533–554, 2020. doi:10.1007/s10817-019-09516-0.

18 E. Jeřábek. Dual weak pigeonhole principle, Boolean complexity, and derandomization. *Annals of Pure and Applied Logic*, 129:1–37, 2004. doi:10.1016/j.apal.2003.12.003.

19 E. Jeřábek. The strength of sharply bounded induction. *Mathematical Logic Quarterly*, 52:613–624, 2006. doi:10.1002/malq.200610019.

20 D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988. doi:10.1016/0022-0000(88)90046-3.

21 B. Kiesl, A. Rebola-Pardo, and M. J. H. Heule. Extended resolution simulates DRAT. In *International Joint Conference on Automated Reasoning (IJCAR '18)*, pages 516–531, 2018. doi:10.1007/978-3-319-94205-6_34.

22 Leszek Aleksander Kołodziejczyk and Neil Thapen. The strength of the dominance rule, 2024. arXiv:2406.13657.

23 J. Krajíček. *Proof complexity*, volume 170 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge, 2019.

24 J. Krajíček and P. Pudlák. Quantified propositional calculi and fragments of bounded arithmetic. *Z. Math. Logik Grundlag. Math.*, 36(1):29–46, 1990. doi:10.1002/malq.19900360106.

25 O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

26 J. P. Marques-Silva and K. A. Sakallah. GRASP-a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999. doi:10.1109/12.769433.

27 J. B. Paris and A. J. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 317–340. Springer-Verlag, 1985. doi:10.1007/BFb0075316.

28 A. A. Razborov. Propositional proof complexity. In *European Congress of Mathematics 2021*, pages 439–464. EMS Press, 2023.

29 S. Riis. Finitisation in bounded arithmetic. *BRICS Report Series*, 23, 1994.

30 N. Wetzler, M. J. H. Heule, and W. A. Hunt, Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing (SAT '14)*, pages 422–429, 2014. doi:10.1007/978-3-319-09284-3_31.

31 A. J. Wilkie and J. B. Paris. On the scheme of induction for bounded arithmetic formulas. *Annals of Pure and Applied Logic*, 35:261–302, 1987. doi:10.1016/0168-0072(87)90066-2.

Dynamic Blocked Clause Elimination for Projected Model Counting

Jean-Marie Lagniez
Univ. Artois, CNRS, CRIL, France

Pierre Marquis
Univ. Artois, CNRS, CRIL, IUF, France

Armin Biere
University Freiburg, Germany

Abstract

In this paper, we explore the application of blocked clause elimination for projected model counting. This is the problem of determining the number of models $\|\exists X.\Sigma\|$ of a propositional formula Σ after eliminating a given set X of variables existentially. Although blocked clause elimination is a well-known technique for SAT solving, its direct application to model counting is challenging as in general it changes the number of models. However, we demonstrate, by focusing on projected variables during the blocked clause search, that blocked clause elimination can be leveraged while preserving the correct model count. To take advantage of blocked clause elimination in an efficient way during model counting, a novel data structure and associated algorithms are introduced. Our proposed approach is implemented in the model counter `d4`. Our experiments demonstrate the computational benefits of our new method of blocked clause elimination for projected model counting.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Projected model counting, blocked clause elimination, propositional logic

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.21

Supplementary Material *Software*: `https://zenodo.org/records/11854002`

Funding This work has benefited from the support of the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) of the French National Research Agency.

1 Introduction

Propositional model counting consists determines the number of models of a propositional formula Σ, typically represented in conjunctive normal form (`CNF`). Many applications however require a projected variant focusing on a specific set X of variables of interest: given a propositional formula Σ and a set X of propositional variables to be forgotten, the projected model counting problem consists in computing the number of interpretations over the variables occurring in Σ but not in X, which coincide on X with a model of Σ. In other words, the goal is to count the number of models of the quantified Boolean formula $\exists X.\Sigma$ over its variables (i.e., those present in Σ but absent in X).

The projected model counting problem is significant for various application domains of artificial intelligence (AI). For instance, in planning scenarios, it helps to evaluate the robustness of a plan by determining the number of initial states from which the plan execution leads to a goal state [4]. Additionally, its applicability extends beyond AI to formal verification problems [19] and database operations [1]. As a generalization of the standard model counting problem `#SAT`, for the special case $X = \emptyset$, the projected model counting problem is at least as complex as `#SAT` (`#P-hard`). However, the possibility to eliminate some variables actually also might simplify the problem, such as when all variables in Σ belong to X, reducing the problem to simply determining the satisfiability of Σ. However, in practice projected model counting often turns out to be more challenging than the standard model counting problem. This can be explained by the additional constraints imposed on the branching heuristic, i.e., in which order variables can be used as decisions, making the problem inherently harder. This is also reflected for instance in upper known bounds in the literature [9] on the time complexity of model counting even formulas with fixed treewidth k, i.e., $\mathcal{O}(2^k)$ for standard and $\mathcal{O}(2^{2^k})$ projected model counting.

One way to speed up model counting is to employ preprocessing which simplifies the formula before tackling the model counting task. Preprocessing methods have shown to be effective across various automated reasoning tasks, notably in SAT solving and QBF solving [6]. Among these preprocessing techniques, blocked-clause elimination (BCE) [15] significantly improves solver performance by emulating several other, more complex preprocessing techniques [16]. Blocked clauses, initially introduced by Kullmann [20] as a generalization of extended resolution clauses, are pivotal in propositional preprocessing techniques. In essence, a clause α is deemed blocked within a CNF formula Σ if it includes a literal ℓ for which all conceivable resolvents of α over ℓ yield tautologies. Removal of blocked clauses can significantly enhance the performance of SAT solvers [15]. Furthermore, generalized forms of BCE have demonstrated remarkable performance improvements in solving problems beyond NP, such as QBF [12], DQBF [31] and even first-order theorem proving [18].

However, while several preprocessing techniques used for SAT solving can be adapted to improve model counting [25, 22], others, such as the blocked clause elimination technique, are unsuitable due to their inability to preserve the number of models. In this paper, we address this challenge by delineating conditions under which the use of BCE is correct in projected model counting. Specifically, we demonstrate that focusing on projected variables during the blocked clause search is correct, i.e., gives the same projected count. The rationale behind this lies in the fact that when concentrating on sub-formulas containing only projected variables, the requirement boils down to ensure satisfiability. Consequently, clauses blocked on projected variables can safely be removed.

When used for model counting, simplification techniques are typically applied up-front during preprocessing and even though modern SAT solvers make heavily use of interleaving formula simplification with CDCL search, also called *inprocesssing* [17], this form of simplification is currently performed only at the root-level (decision level zero). In this paper we go beyond root-level simplification and propose to dynamically apply the blocked clause elimination technique dynamically during search at every decision level in the form of *dynamic blocked clause elimination.* In this sense our approach is similar to look-ahead solving [14], which use simplification techniques during search, i.e., probing techniques, at every decision level.

To accomplish this, we introduce novel data structures and associated algorithms tailored for dynamic inprocessing. Our method efficiently identifies clauses eligible for elimination by employing a mechanism akin to watched literals. Importantly, this methodology is not tied to a specific model counter; it seamlessly integrates into any state-of-the-art model counter.

To assess the efficiency of our approach, we conducted experiments using the model counter d4 [24], modified to integrate our newly developed data structures and algorithms for projected model counting. We evaluated the performance of this new version of d4 across various benchmarks from recent model counting competitions (available at https://mccompetition.org/). Our experimental results underscore the computational advantages of employing blocked clause elimination for projected model counting. For certain benchmarks, the adoption of BCE dynamic inprocessing led to a substantial reduction in computation time, with time savings of up to one order of magnitude compared to the baseline version of d4. To ensure that the improvements are indeed attributable to the use of BCE inprocessing, we also examined a version of d4 that implements BCE dynamically during preprocessing only.

Interestingly, our findings indicate that employing BCE in preprocessing had no discernible impact on the effectiveness of the model counter d4, underscoring the possibility to take advantage of BCE eagerly during the model counting process.

The remainder of the paper is structured as follows. The next section provides formal preliminaries. Following this, we delve into theoretical insights and give implementation details on how to perform BCE dynamically during search. Then, we outline the experimental protocol adopted for our empirical evaluations, along with the corresponding results. Finally, we conclude the paper, offering insights into potential avenues for future research. The source code and benchmarks utilized in our experiments are provided as supplementary materials.

2 Preliminaries

Let $\mathcal{L}$ be a propositional language built up from a finite set of propositional variables $\mathcal{P}$ and the standard logical connectives. The symbols $\bot$ and $\top$ represent the Boolean constants for falsehood and truth, respectively. A *literal* ℓ is either a propositional variable (e.g., x) or its negation ($\neg x$). For a literal ℓ defined over variable x, its *complementary literal* $\overline{\ell}$ is defined as $\overline{\ell} = \neg x$ if $\ell = x$, and $\overline{\ell} = x$ if $\ell = \neg x$, with $Var(\ell) = x$ denoting the variable of ℓ. A *term* is a conjunction of literals. A *clause* is a disjunction of literals. Terms and clauses are also interpreted as their sets of literals whenever convenient.

A clause is a *tautology* if it contains $\top$, or both x and $\neg x$ for some variable x. A CNF formula Σ is a conjunction of clauses, also viewed as set of clauses when needed. The set of propositional variables occurring in Σ is denoted $Var(\Sigma)$. If a variable $x \in X$ does not belong to $Var(\Sigma)$, then x is said to be *free* in Σ. Each clause is associated with a unique identifier represented as an integer. A clause α_i of a CNF formula Σ can be accessed using its identifier through square bracket notation, denoted as $\Sigma[i]$. Thus α_i is also noted $\Sigma[i]$. We denote by $S_\ell(\Sigma)$ the set of clauses of Σ that contain literal ℓ. When no ambiguity about Σ is possible, we simply use the shorthand notation S_ℓ instead of explicitly writing $S_\ell(\Sigma)$.

▶ **Example 1.** Consider the CNF formula $\Sigma = \{\alpha_1, \alpha_2, \ldots, \alpha_{11}\}$ with

$1 : x_1 \vee x_2$	$2 : \neg x_2 \vee x_3$	$3 : \neg x_1 \vee \neg x_2 \vee \neg y_1$	$4 : x_1 \vee \neg x_3 \vee y_1$
$5 : x_2 \vee \neg x_3 \vee y_2$	$6 : x_1 \vee \neg x_3 \vee \neg y_2$	$7 : y_3 \vee x_2$	$8 : \neg y_3 \vee \neg x_2 \vee \neg x_3$
$9 : \neg y_3 \vee x_1$	$10 : \neg y_3 \vee \neg y_2 \vee x_3$	$11 : y_3 \vee y_2 \vee x_2$	

$Var(\Sigma) = \{x_1, x_2, x_3, y_1, y_2, y_3\}$, $S_{x_1}(\Sigma) = \{\alpha_1, \alpha_4, \alpha_6, \alpha_9\}$, and $\Sigma[2] = \alpha_2 = \neg x_2 \vee x_3$.

An *interpretation* (or world) over $\mathcal{P}$, denoted by ω, is a mapping from $\mathcal{P}$ to $\{0, 1\}$. Interpretations ω are often represented by sets of literals (one per variable in $\mathcal{P}$), of exactly those literals set to 1 by ω. The collection of all interpretations is denoted by $\mathcal{W}$. An interpretation ω is a *model* of a formula $\Sigma \in \mathcal{L}$ if and only if it satisfies the formula in accordance with the usual truth-functional interpretation. The set of models of the formula Σ is denoted by $\text{mod}(\Sigma)$, defined as $\{\omega \in \mathcal{W} \mid \omega \models \Sigma\}$. The symbol $\models$ denotes logical entailment, while $\equiv$ denotes logical equivalence. For any formulas $\Sigma, \Psi \in \mathcal{L}$, we have $\Sigma \models \Psi$ if and only if $\text{mod}(\Sigma) \subseteq \text{mod}(\Psi)$ and $\Sigma \equiv \Psi$ if and only if $\text{mod}(\Sigma) = \text{mod}(\Psi)$. The notation $\|\Sigma\|$ indicates the number of models of Σ over $Var(\Sigma)$.

▶ **Example 2** (Example 1 cont'd). $\|\Sigma\| = 9$ and the models of Σ are:

$\{\neg x_1, x_2, x_3, y_1, \neg y_2, \neg y_3\}$	$\{x_1, \neg x_2, x_3, y_1, y_2, y_3\}$	$\{x_1, \neg x_2, x_3, \neg y_1, y_2, y_3\}$
$\{x_1, \neg x_2, \neg x_3, \neg y_1, y_2, y_3\}$	$\{x_1, \neg x_2, \neg x_3, y_1, \neg y_2, y_3\}$	$\{x_1, \neg x_2, \neg x_3, y_1, y_2, y_3\}$
$\{x_1, \neg x_2, \neg x_3, \neg y_1, \neg y_2, y_3\}$	$\{x_1, x_2, x_3, \neg y_1, y_2, \neg y_3\}$	$\{x_1, x_2, x_3, \neg y_1, \neg y_2, \neg y_3\}$

For a formula $\Sigma \in \mathcal{L}$ and a subset $X \subseteq \mathcal{P}$, $\exists X.\Sigma$ represents, up to logical equivalence, the most general consequence of Σ that is independent of the variables in X (see for instance [28] for details). We note $Var(\exists X.\Sigma) = Var(\Sigma) \setminus X$.

▶ **Example 3** (Example 1 cont'd). Let $X = \{y_1, y_2, y_3\}$. We have $\|\exists X.\Sigma\| = 4$ and the models of $\exists X.\Sigma$ over $Var(\exists X.\Sigma)$ are $\{\{\neg x_1, x_2, x_3\}, \{x_1, \neg x_2, x_3\}, \{x_1, \neg x_2, \neg x_3\}, \{x_1, x_2, x_3\}\}$.

The *conditioning* of a `CNF` formula Σ by a consistent term γ results in the formula denoted by $\Sigma_{|\gamma}$, where $\Sigma_{|\gamma}$ is obtained from Σ by removing each clause from containing a literal of γ and simplifying the remaining clauses, by removing from them complementary literals to those in γ. If, during the simplification, a clause becomes empty, then $\Sigma_{|\gamma}$ is unsatisfiable.

The conditioning of Σ on ℓ is equivalent to the formula $\exists\, Var(\ell).(\Sigma \wedge \ell)$. When ℓ is a unit clause of Σ, $\Sigma_{|\ell}$ is satisfiable if and only if Σ is satisfiable. *Boolean Constraint Propagation* (`BCP`) [29] is the algorithm that, given a `CNF` formula Σ, returns a `CNF` formula closed under unit propagation, i.e., that does not contain any unit clauses. The resulting formula is obtained by repeating the unit propagation of a unit clause of Σ in the formula Σ while such a unit clause exists. The identifiers assigned to clauses in Σ remain unaltered by `BCP`. Consequently, $\texttt{BCP}(\Sigma)[i]$ will retrieve the clause α_i resulting from the application of `BCP` on Σ, which could be $\bot$, $\top$, or a subset of α_i.

▶ **Example 4** (Example 1 cont'd). The formula $\texttt{BCP}(\Sigma_{|\neg x_1}) = (x_2) \wedge (x_3) \wedge (y_1) \wedge (\neg y_2) \wedge (\neg y_3)$ is the result of conditioning Σ with the literal $\neg x_1$ and applying `BCP` to $\Sigma_{|\neg x_1}$. $\texttt{BCP}(\Sigma_{|\neg x_1})[1] = \top$ and $\texttt{BCP}(\Sigma_{|\neg x_1})[4] = y_1$.

The resolution rule asserts that given two clauses $\alpha_1 = \{\ell, a_1, \ldots, a_n\}$ and $\alpha_2 = \{\overline{\ell}, b_1, \ldots, b_m\}$, the resulting clause $\alpha = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, is the *resolvent* of α_1 and α_2 on the literal ℓ. This operation is denoted as $\alpha = \alpha_1 \oplus \alpha_2$. This concept extends naturally to sets of clauses: for two sets S_ℓ and $S_{\overline{\ell}}$ containing clauses that all involve ℓ and $\overline{\ell}$, respectively, we define $S_\ell \oplus S_{\overline{\ell}} = \{\alpha_1 \oplus \alpha_2 | \alpha_1 \in S_\ell, \alpha_2 \in S_{\overline{\ell}}, \text{ and } \alpha_1 \oplus \alpha_2 \text{ is not a tautology}\}$.

▶ **Example 5** (Example 1 cont'd). Let $S_{\neg x_1} = \{(\neg x_1 \vee \neg x_2 \vee \neg y_1)\}$, $S_{y_3} = \{(y_3 \vee x_2), (y_3 \vee y_2 \vee x_2)\}$ and $S_{\neg y_3} = \{(\neg y_3 \vee \neg x_2 \vee \neg x_3), (\neg y_3 \vee x_1), (\neg y_3 \vee \neg y_2 \vee x_3)\}$. We have $S_{y_3} \oplus S_{\neg y_3} = \{(x_1 \vee x_2), (\neg y_2 \vee x_3 \vee x_2), (y_2 \vee x_2 \vee x_1)\}$ and $\{(x_1 \vee x_2)\} \oplus S_{x_1} = \emptyset$.

The simplification technique known as *Blocked Clause Elimination* (`BCE`) [15, 12], targets the removal of specific clauses termed *blocked clauses* from `CNF` formulas [20]. In the context of a `CNF` formula Σ, a literal ℓ within a clause α is termed a *blocking literal* if it blocks α with respect to Σ. This occurs when, for every clause α' in Σ containing $\overline{\ell}$, the resulting resolvent $\alpha \oplus \alpha'$ on ℓ is a tautology. In essence, for a given `CNF` and its clauses, a clause is considered blocked if it contains a literal that can effectively block it. Applying `BCE` to Σ leads to remove every clause containing a blocking literal and by repeating the process iteratively until no blocked literal exists. [15, 12] illustrates that the outcome of `BCE` remains satisfiable equivalent regardless of the sequence in which blocked clauses are eliminated. More generally, blocked clause elimination converges to a unique fixed point for any `CNF` formula, establishing the confluence of the method.

▶ **Example 6** (Example 1 cont'd). Above we have shown that the clause $(x_1 \vee x_2)$ is blocked by x_1 and therefore can be eliminated. Following this, both $(\neg x_1 \vee \neg x_2 \vee \neg y_1)$ and $(x_1 \vee \neg x_3 \vee y_1)$ can be removed interchangeably, as they are respectively blocked by y_1 and $\neg y_1$. Subsequently, $(\neg y_3 \vee \neg y_2 \vee x_3)$, blocked by $\neg y_2$, is eliminated, along with the two clauses, $(\neg y_3 \vee x_1)$ and $(x_1 \vee \neg x_3 \vee \neg y_2)$, both blocked by x_1. Next, $(y_3 \vee x_2)$ is removed as it is blocked by y_3. Following this, both $(x_2 \vee \neg x_3 \vee y_2)$ and $(y_3 \vee y_2 \vee x_2)$, blocked by y_2, can be eliminated. Then, $(\neg y_3 \vee \neg x_2 \vee \neg x_3)$ is removed because it is blocked by y_3, and finally, the last clause $(\neg x_2 \vee x_3)$ is removed because it is blocked by both $\neg x_2$ and x_3. Thus, $\texttt{BCE}(\Sigma) = \emptyset$.

As highlighted in [20], the removal of any blocked clause ensures the preservation of unsatisfiability. However, as illustrated by the previous example, utilizing blocked clause elimination (`BCE`) on a `CNF` formula Σ does not ensure that the resulting formula `BCE`(Σ) has the same number of models as Σ. In the next section, we will delve into the specific conditions under which `BCE` can be effectively used for projected model counting.

3 Blocked Clause Elimination for Projected Model Counting

Our goal is to use blocked clause elimination dynamically during search in projected model counting. The primary challenge is to identify conditions under which such simplification is allowed. Section 3.1 provides novel theoretical insights permitting the removal of blocked clauses and Section 3.2 introduces new algorithms to efficiently identify them.

3.1 Theoretical Insights

As illustrated by Example 6, the `BCE` rule cannot be applied indiscriminately. When applied to the formula Σ provided in Example 1, the result is a tautological formula, indicating that $\|\texttt{BCE}(\Sigma)\| = 1$ (since $Var(\texttt{BCE}(\Sigma)) = \emptyset$ this correspond to $2^6 = 64$ models over $Var(\Sigma)$), which differs from $\|\Sigma\| = 9$. It is essential to note that blocked clause elimination guarantees the preservation of satisfiability but not necessarily equivalence or the number of models. However, the picture changes when addressing the projected model counting problem. As we will demonstrate in Proposition 7, it is feasible to eliminate clauses that are blocked on projected variables. The rationale behind this lies in the fact that when focusing on sub-formulas containing only projected variables, the requirement is only to ensure satisfiability. Consequently, clauses blocked on projected variables can be removed in this case:

▶ **Proposition 7.** *Let $\exists x.\Sigma$ be an existentially quantified* `CNF` *formula. If a non-tautological clause $\alpha \in \Sigma$ is blocked by a literal $\ell \in \alpha$ with $Var(\ell) = x$, then $\exists x.\Sigma$ is logically equivalent to $\exists x.\Sigma'$, where $\Sigma' = \Sigma \setminus \{\alpha\}$.*

Proof. To establish the logical equivalence $\exists x.\Sigma \equiv \exists x.\Sigma'$, we need to demonstrate both (1) $\exists x.\Sigma \models \exists x.\Sigma'$ and (2) $\exists x.\Sigma' \models \exists x.\Sigma$. For condition (1) since $\Sigma \models \Sigma'$ it follows directly that $\exists x.\Sigma \models \exists x.\Sigma'$. Now, let us demonstrate the second condition. We have to prove for any interpretation ω satisfying $\exists x.\Sigma'$, that ω also satisfies $\exists x.\Sigma$. Consider an interpretation ω satisfying $\exists x.\Sigma'$. This means that ω satisfies $(\Sigma'_{|x} \vee \Sigma'_{|\neg x})$. We need to address two scenarios depending on whether ω satisfies $\Sigma'_{|x}$ or $\Sigma'_{|\neg x}$. If ω satisfies $\Sigma'_{|x}$, then $\Sigma'_{|x} \equiv \Sigma_{|x}$. Since $\Sigma_{|x}$ entails $\Sigma_{|x} \vee \Sigma_{|\neg x}$, we conclude that ω satisfies $\exists x.\Sigma$. Let us consider the second scenario where ω satisfies $\Sigma'_{|\neg x}$ but not $\Sigma'_{|x}$ (the case when $\omega \models \Sigma'_{|x}$ has just been discussed). First, both $\Sigma'_{|x}$ and $\Sigma'_{|\neg x}$ contain clauses from Σ' that do not involve variable x. Therefore, if ω does not satisfy Σ'_x but satisfies $\Sigma'_{\neg x}$, there must be a clause $\beta \in \Sigma'$ with $\neg x \in \beta$ and $\omega \not\models \beta_{|x}$. Now, let us demonstrate that ω satisfies $\Sigma_{|\neg x}$. Since $\Sigma_{|\neg x} \equiv (\Sigma' \wedge \alpha)_{|\neg x} \equiv \Sigma'_{|\neg x} \wedge \alpha_{|\neg x}$, we only need to show that ω satisfies $\alpha_{|\neg x}$. As α is blocked on x in Σ, each resolvent between α and a clause of Σ containing $\neg x$ is a tautology. Particularly, $\beta \oplus \alpha$ is a tautology, implying that there exists a literal $\exists y \in \beta$ such that $\neg y \in \alpha$ and $x \neq y$. Since we have established that $\omega \not\models \beta_{|x}$, this implies that ω satisfies $\neg y$, hence ω satisfies $\alpha_{|\neg x}$. This demonstrates that ω satisfies $\Sigma'_{\neg x} \wedge \alpha_{\neg x}$, and consequently, ω satisfies $\Sigma_{|\neg x}$. Using similar reasoning as before, we can show that ω satisfies $\exists x.\Sigma$. Therefore, for any interpretation ω that satisfies $\exists x.\Sigma'$, it follows that ω satisfies $\exists x.\Sigma$, proving $\exists x.\Sigma' \models \exists x.\Sigma$. ◀

Proposition 7 only considers formulas with a single existentially quantified and thus projected variable. This can be extended to sets of variables:

▶ **Corollary 8.** *Let $\exists X.\Sigma$ be an existentially quantified* CNF *formula. If a non-tautological clause $\alpha \in \Sigma$ is blocked by a literal $\ell \in \alpha$ such that $Var(\ell) \in X$, then $\exists X.\Sigma$ is logically equivalent to $\exists X.\Sigma'$, where $\Sigma' = \Sigma \setminus \{\alpha\}$.*

Proof. The proof is straightforward. Proposition 7 establishes $\exists x.\Sigma \equiv \exists x.\Sigma'$. Therefore, we directly deduce that $\exists X \setminus \{x\}.(\exists x.\Sigma) \equiv \exists X \setminus \{x\}.(\exists x.\Sigma')$. ◀

Corollary 8 demonstrates the potential of utilizing blocked clause elimination to enhance projected model counters. Our objective is not only to identify the set of blocked clauses in preprocessing but also to perform this operation during search dynamically. However, naive algorithms for blocked clause elimination are in the worst case at least quadratic in the size of the formula, which is clearly infeasible for dynamic blocked clause elimination. In the following section, we capitalize on the observation that model counters typically follow the trace of DPLL solvers. To efficiently detect blocking literals and remove blocked clauses, a dedicated data structure along with associated algorithms are designed.

3.2 Implementation Details

To improve the efficiency of identifying clauses eligible for removal through the blocked clause elimination rule, we introduce the `BlockedClauseManager` object in this section. This specialized utility integrates efficient structures and algorithms crafted for this purpose, and is not exclusive to the projected model counter `d4`. It can be seamlessly employed in any state-of-the-art projected model counter.

To identify clauses eligible for elimination due to being blocked by a literal, we use a mechanism akin to the concept of watched literals. Given a formula $\exists X.\Sigma$, we aim to capture scenarios where a clause α cannot be eliminated via the blocked clause elimination rule, which occurs when there is no literal $\ell \in \alpha$ such that α is blocked on x, and $Var(\ell) \in X$. Specifically, a clause α is not blocked on a literal $\bar{\ell} \in \alpha$ if there exists another clause α' such that $\bar{\ell} \in \alpha'$ and $\alpha \oplus \alpha'$ is not a tautology. Consequently, the invariant we adopt stipulates that for each literal $\ell \in \alpha$ such that $Var(\ell) \in X$, either ℓ is assigned or there must exist a clause α' where $\bar{\ell} \in \alpha'$, and $\alpha \oplus \alpha' \not\equiv \top$.

▶ **Example 9** (Example 1 cont'd)**.** When evaluating α_3, it is not feasible to associate the literal $\neg y_1$ with a clause from Σ without resulting in a tautology. Therefore, α_3 can be safely removed from Σ. Conversely, when examining α_{11}, it is feasible to associate the literal y_3 with clause α_9 and the literal y_2 with clause α_6, demonstrating that α_{11} cannot be eliminated from the formula using the blocked clause elimination rule.

Since blocked elimination can ignore (implied) learned clauses [17], the set $\{\alpha\} \oplus S_{\bar{\ell}}$, representing possible resolutions on a literal ℓ concerning a clause $\alpha \in \Sigma$, can be computed once at the outset. Consequently, when the watched clause to assess whether α is blocked on ℓ is deactivated, it suffices to consider clauses in $\{\alpha\} \oplus S_{\bar{\ell}}$ rather than re-evaluating each clause of $S_{\bar{\ell}}$ to determine if the resolution rule yields a tautology. The first data structure incorporated into our `BlockedClauseManager` is thus a set of triples $(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}})$, referred to as `protectedTriple`.

The function `initProtectedTriple`, outlined in Algorithm 1, is designed for this purpose. When provided with the existentially quantified CNF formula $\exists X.\Sigma$, it begins by enumerating all variables x in X (lines 2–5). Subsequently, it iterates through each possible triple $(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}})$ such that $\ell \in \{x, \neg x\}$, $\ell \in \alpha$, and $\alpha \in S_\ell$ (lines 3–5), adding them into `protectedTriple` (line 5). Moving forward, we will primarily work with clause identifiers rather than the clauses themselves. Therefore, when referring to a clause α in the following discussions, we are actually addressing its identifier. This applies similarly to sets of clauses; we will focus on the set of identifiers corresponding to the clauses within the set.

Algorithm 1 `initProtectedTriple`.

```
Input: $\exists X.\Sigma$ an existentially quantified CNF formula.

1 protectedTriple ← $\emptyset$
2 for $x \in X$ do
3     for $\ell \in \{x, \neg x\}$ do
4         for $\alpha \in S_\ell(\Sigma)$ do
5             protectedTriple ← protectedTriple $\cup \{(\ell, \alpha, \{\alpha\} \oplus S_{\bar{\ell}}(\Sigma))\}$
```

▶ **Example 10** (Example 1 cont'd). Upon invoking the function `initProtectedTriple` on the existentially quantified formula $\exists X.\Sigma$ provided in Example 1, the set `protectedTriple` contains the following triples: $(y_1, 4, \{\})$, $(\neg y_1, 3, \{\})$, $(y_2, 5, \{6\})$, $(y_2, 11, \{6\})$, $(\neg y_2, 10, \{\})$, $(\neg y_2, 6, \{5, 11\})$, $(y_3, 7, \{9, 10\})$, $(y_3, 11, \{9\})$, $(\neg y_3, 8, \{\})$, $(\neg y_3, 9, \{7, 11\})$, $(\neg y_3, 10, \{7\})$.

For each triple $(\ell, \alpha, \mathcal{C})$ in `protectedTriple`, we need to watch a clause from $\mathcal{C}$ to ensure that clause α is not blocked by ℓ. To achieve this, we incorporate into `BlockedClauseManager` a map of watching lists, denoted as `watches`. This structure associates each clause $\alpha \in \Sigma$ with a set of triples `watches[`α`]` that are being watched by α.

Algorithm 2 presents the pseudo-code for the function `initWatchList`. Given an existentially quantified CNF formula $\exists X.\Sigma$, this function initializes the `watches` structure and returns the indices of blocked clauses U, which are the clauses for which it is impossible to associate a sentinel. The function begins by initializing the set of blocked clauses as empty (line 2). Then, it initializes the map `watches` by associating an empty set with each clause of Σ (lines 2–3). Next, it iterates over the triples in the `protectedTriple` set to associate a sentinel with each of them (lines 4–6). For each triple $t = (\ell, \alpha, \mathcal{C})$, where $\mathcal{C}$ represents the set of non-tautological clauses, the algorithm checks whether $\mathcal{C}$ is empty. If it is, α is added to the set of blocked clauses (line 5). Otherwise, a clause α' from $\mathcal{C}$ is selected, and the triple t is added to the watching list of α' (line 6).

Algorithm 2 `initWatchList`.

```
Input: $\exists X.\Sigma$ an existentially quantified CNF formula.
Output: $B$ is a set of identifiers of clauses that are blocked.

1 $U \leftarrow \emptyset$
2 Let watches an empty map
3 for $\alpha \in \Sigma$ s.t. $Var(\alpha) \cap X \neq \emptyset$ do  watches[$\alpha$] = {}
4 for $t = (\ell, \alpha, \mathcal{C}) \in$ protectedTriple do
5     if $\mathcal{C} = \emptyset$ then  $U \leftarrow U \cup \{\alpha\}$
6     else  watches[$\alpha'$] ← watches[$\alpha'$] $\cup \{t\}$ with $\alpha' \in \mathcal{C}$
7 return U
```

▶ **Example 11** (Example 1 cont'd). Upon calling the function `initWatchList` on the existentially quantified formula $\exists X.\Sigma$ provided in Example 1, the following represents a potential initialization of the `watched` structure:

`watches`$[6] = \{(y_2, 5, \{6\}), (y_2, 11, \{6\})\}$ `watches`$[5] = \{(\neg y_2, 6, \{5, 11\})\}$

`watches`$[9] = \{(y_3, 7, \{9, 10\}), (y_3, 11, \{9\})\}$ `watches`$[7] = \{(\neg y_3, 9, \{7, 11\}), (\neg y_3, 10, \{7\})\}$

`watches`$[3] =$ `watches`$[4] =$ `watches`$[8] =$ `watches`$[10] =$ `watches`$[11] = \emptyset$

To finalize the initialization of the `BlockedClauseManager` object, we incorporate two arrays for maintaining records of assigned variables and satisfied clauses. The first array, named `isAssignedVar`, associates each variable in X with a Boolean value set to `true` if the variable is assigned, and `false` otherwise. The second array, named `isActiveClause`, associates each clause of Σ (identified by their identifier) with a Boolean variable set to `true` if the clause is active, and `false` otherwise. The arrays `isAssignedVar` and `isActiveClause` are initialized with `false` and `true`, respectively, for all their elements. We also need a stack S of pairs, each consisting of variables and clauses. This stack is used to track the changes made to `isAssignedVar` and `isActiveClause` during each call of the function `propagate`.

Algorithm 3 outlines all the necessary steps for the initialization process. It begins by initializing the two arrays (lines 1–2). Then, the set of triples is initialized by invoking the function `initProtectedTriple` on $\exists X.\Sigma$. Next, the `watches` structure is initialized by calling the function `initWatchList` on $\exists X.\Sigma$, and the set of blocked clauses is collected in U. We initialize S as an empty stack of pairs, where each pair consists of a set of variables and a set of clauses. Finally, the function `propagate` is called to gather all the initially blocked clauses. This function, described afterwards, takes a set of inactive clauses and a set of freshly assigned variables as input, and returns a set of clauses that are identified as blocked (further details will be provided later).

Algorithm 3 `init`.

Input: $\exists X.\Sigma$ an existentially quantified `CNF` formula.
Output: B is a set of identifiers of clauses that are blocked.

```
1 Let isAssigned be an array s.t. isAssigned[$x$] = false for each $x \in X$
2 Let isActiveClause be an array s.t. isActiveClause[$\alpha$] = true for each $\alpha \in \Sigma$
3 initProtectedTriple($\exists X.\Sigma$)
4 $U \leftarrow$ initWatchList($\exists X.\Sigma$)
5 $S$ is an empty stack of pairs of the form (variables, clauses)
6 return propagate($U$, $\emptyset$)
```

Before delving into the specifics of how the function `propagate` operates, it is important to highlight that when conditioning a formula by a literal x, without rendering it unsatisfiable, there is no need to consider clauses shortened by this assignment. Consider a clause $\alpha \in \Sigma$ with $\bar{\ell} \in \alpha$. We aim to demonstrate that $\alpha \setminus \{\bar{\ell}\} \in \Sigma_{|\ell}$ cannot be blocked by any literal $\ell' \in \alpha \setminus \{\ell\}$ in $\Sigma_{|\ell}$. Given that α is not blocked in Σ, for every $\ell' \in \alpha \setminus \{\bar{\ell}\}$, there exists $\alpha' \in \Sigma$ such that $\bar{\ell'} \in \alpha'$ and $\alpha \oplus \alpha' \not\equiv \top$. Firstly, note that $\ell \notin \alpha'$; otherwise, $\alpha \oplus \alpha'$ would be a tautology. We then consider two cases based on whether $\bar{\ell}$ belongs to α'. In the first case when $\bar{\ell} \in \alpha'$, we have $\alpha' \setminus \{\bar{\ell}\} \in \Sigma_{|\ell}$, and since the resolution between $\alpha \setminus \{\bar{\ell}\}$ and $\alpha' \setminus \{\bar{\ell}\}$ is not a tautology, it follows that $\alpha \setminus \{\ell\}$ is not blocked on ℓ'. In the second case where $\bar{\ell} \notin \alpha'$, $\alpha' \in \Sigma_{|\ell}$, and once more, $\alpha \setminus \{\ell\}$ is not blocked on ℓ' because $\alpha' \setminus \{\ell\} \oplus \alpha' \not\equiv \top$.

▶ **Example 12** (Example 9 cont'd). Let us examine the formula $\Sigma_{|\neg x_2}$. It is evident that α_{11} remains an unblocked because, with the literals present in the resulting clauses $y_3 \vee y_2$, we can still reference the same clauses from $\Sigma_{|\neg x_2}$ to maintain the invariant.

There are two scenarios where it becomes pertinent to evaluate whether a clause can be eliminated due to being blocked: when an active clause has been satisfied by a literal, or when an active clause has been blocked by a literal. Thus, once certain clauses become inactive, that are clauses satisfied or blocked, it becomes imperative to update the `watches` structure accordingly. The process for this update closely resembles the mechanism for updating watched literals in modern SAT solvers. Specifically, for each newly inactive clause α, we need to iterate through the list of triples from $\texttt{watches}[\alpha]$ associated with α. For each triple $t = (\ell, \alpha, \mathcal{C})$ of $\texttt{watches}[\alpha]$, if ℓ is not assigned and α is active, we must search for another sentinel in $\mathcal{C}$ – that is, an active clause. The concept here is to ensure that each triple $t = (\ell, \alpha, \mathcal{C})$ is linked with a clause. Additionally, if t is active, meaning α is active, it should be watched by an active clause. Otherwise, if t is watched by an inactive clause α', we must ensure that when we reactivate α, α' is also made active again. This aspect is crucial as it guarantees the backtrack freeness of our structure.

Algorithm 4 outlines the pseudo-code for the function `propagate`, which fulfills the aforementioned requirements. It takes as input a set of clauses identified by their identifiers, denoted as U, that have become inactive, and a set of freshly assigned variables Y. The function returns the set B of clauses detected as being blocked. The algorithm begins by updating the two arrays `isAssignedVar` and `isActiveClause` to reflect the assignment of variables in Y and the inactivation of clauses in U (lines 1–2). Next, the set of blocked clauses B is initialized to be empty (line 3). Then, for each inactive clause α stored in U, the `watches` map is updated (lines 4–18). To accomplish this, an inactive clause α is selected and removed from U (lines 5–6). Subsequently, the set `tmpWatch` is initialized to be empty, and it is used to store triples that will be watched by α, containing triples with assigned variables or inactive clauses (line 18). Since α is now inactive, active triples associated with α need to be redistributed to other active clauses.

This operation is conducted in the for loop where each triple $t = (\ell, \alpha', \mathcal{C})$ from $\texttt{watches}[\alpha]$ is considered (lines 8–17). If ℓ is assigned or if α' is inactive (line 9), α can continue to watch t, and thus t is added to `tmpWatch` (line 10). However, if there exists an active clause α'' in $\mathcal{C}$ (line 11), then triple t is added to the watch list of α'' (line 12). Lastly, if it is impossible to associate t with an active clause (lines 13–17), then clause α' is considered blocked, implying that α can continue to watch α' since they will both become active together upon backtracking (line 14). Moreover, α' is added to B, α' is added to U to handle α' later (line 16), and clause α' is marked as inactive (line 17). Upon completing the update of the `watches` map, the assigned variables and inactivated clauses are pushed onto stack S (line 19). Finally, the set of clauses identified as blocked is returned at line 20.

▶ **Example 13** (Example 1 cont'd)**.** Consider the scenario where the literal x_1 is assigned to `true`. In this case, clauses $1, 4, 6, 9$ become satisfied. Invoking the function `propagate` with this information will result in the function returning $\{11\}$ as the set of detected blocked clauses. The various structures within our `BlockedClauseManager` object will be updated as follows:

$\texttt{watches}[7] = \{(\neg y_3, 9, \{7, 11\}), (\neg y_3, 10, \{7\})\}$ $\texttt{watches}[5] = \{(\neg y_2, 6, \{5, 11\}), (\neg y_2, 10, \{5\})\}$
$\texttt{watches}[9] = \{(y_3, 11, \{9\})\}$ $\texttt{watches}[6] = \{((y_2, 11, \{6\})\}$
$\texttt{watches}[10] = \{(y_2, 5, \{6, 10\}), (y_3, 7, \{9, 10\})\}$
$\texttt{watches}[3] = \texttt{watches}[4] = \texttt{watches}[8] = \texttt{watches}[11] = \emptyset$ $S = (\{x_1\}, \{1, 4, 6, 9, 11\})$
$\texttt{isActiveClause} = [0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0]$ $\texttt{isAssignedVar} = [1, 0, 0]$

As mentioned earlier, our structure is designed to be backtrack-free. Therefore, the only operation needed during backtracking is to retrieve from the stack S the elements of the two arrays `isAssignedVar` and `isActiveClause` that require reinitialization. Algorithm 5 outlines the steps involved in the backtracking process.

To conclude this section, let us illustrate how `BlockedClauseManager` is used within a `DPLL`-style projected model counter, such as the one employed in the model counter `d4` [24]. It is worth noting that our proposed approach can also be applied to other types of projected

Algorithm 4 `propagate`.

```
Input: $U$ represents a set of clause identifiers corresponding to the newly inactive
  clauses, and $Y$ denotes the set of variables that have been newly assigned.
Output: $B$ is a set of identifiers of clauses that are blocked.

for $x \in Y$ do isAssigned[y] = true
for $\alpha \in U$ do isActiveClause[$\alpha$] = false
$B \leftarrow \emptyset$
while $U \neq \emptyset$ do
    Let $\alpha \in U$
    $U \leftarrow U \setminus \{\alpha\}$
    tmpWatch = $\emptyset$
    for $t = (x, \alpha', \mathcal{C}) \in$ watches[$\alpha$] do
        if isAssigned[$x$] or not isActiveClause[$\alpha'$] then
            tmpWatch $\leftarrow$ tmpWatch $\cup \{t\}$
        else if $\exists \alpha'' \in \mathcal{C}$ s.t. isActiveClause[$\alpha'$] then
            watches[$\alpha''$] $\leftarrow$ watches[$\alpha''$] $\cup \{t\}$
        else
            tmpWatch $\leftarrow$ tmpWatch $\cup \{t\}$
            $B \leftarrow B \cup \{\alpha'\}$
            $U \leftarrow U \cup \{\alpha'\}$
            isActiveClause[$\alpha'$] $\leftarrow$ false
    watches[$\alpha$] $\leftarrow$ tmpWatch
push the couple $(Y, B \cup U)$ in $S$
return $B$
```

model counters, such as those discussed in [30, 26, 8, 7, 10]. Algorithm 6 outlines the `count` function, which is invoked on $\exists X.\Sigma$, an existentially quantified `CNF` formula, returning the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$. Specifically, this function creates a global variable named *bce*, which is a `BlockedClauseManager` object (line 1), initializes it (line 2), removes the detected blocked clauses (line 3), and calls the recursive algorithm `count_main` on the simplified formula (line 4). It is important to note that while this function initializes the object necessary for enforcing `BCE` during model counting, the actual computation of the number of models is performed within the `count_main` function, which is described afterwards.

Algorithm 7 outlines the recursive function `count_main`, which serves as a pseudo-code representation of a `DPLL`-style projected model counter. This function operates on $\exists X.\Sigma$, an existentially quantified `CNF` formula, and computes the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$. The blue portion of the algorithm, which differs from the baseline due to the inclusion of `BCE` management, will be discussed later.

The function begins by invoking `BCP` on the input formula Σ at line 1. For simplicity, we assume that `BCP` returns a triple consisting of the set of unit literals *units*, the set of satisfied clauses S, and the simplified formula Σ without clauses from S and the unit literals from *units*. If the formula returned by `BCP` contains an empty clause, indicating unsatisfiability, the function returns 0 (line 3). At line 5, the algorithm visits a cache to determine whether the current formula Σ has been previously encountered during the search. The cache, which

Algorithm 5 `backtrack`.

```
1 $(X, \mathcal{C}) \leftarrow$ top($S$)
2 pop($S$)
3 for $x \in X$ do isAssignedVar[$x$] $=$ false
4 for $\alpha \in \mathcal{C}$ do isActiveClause[$\alpha$] $=$ true
```

Algorithm 6 `count`.

```
Input: $\exists X.\Sigma$ an existentially quantified CNF formula.
Output: the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$
1 global bce is an BlockedClauseManager object
2 $B \leftarrow$ init($bce$)
3 $\Sigma \leftarrow \Sigma \setminus \{\Sigma[i]$ s.t. $i \in B\}$
4 return count_main($\Sigma$)
```

starts empty, stores pairs comprising a CNF formula and its corresponding projected model count with respect to X. Whenever Σ is found in the cache, instead of recalculating $\|\exists X.\Sigma\|$ from scratch, the algorithm retrieves `cache`(Σ) (line 7) to streamline the computation. If the formula is satisfiable, `connectedComponents` is called (line 8) on Σ to partition it into a set of CNF formulae that are pairwise variable-disjoint. This procedure is a standard method employed in model counters. It identifies connected components of the primal graph of Σ and returns a set *comps* of CNF formulae, ensuring that each pair of distinct formulae in *comps* does not share any common variable. The variable *cpt*, used to accumulate intermediate model counts, is initialized to 1 (line 9). Then, the function iterates over the connected components Σ' identified in *comps* (lines 10–14) to count the number of models of each component. If the considered component Σ' only contains variables from X, the model count accumulated in *cpt* is multiplied by 1 if Σ' is satisfiable and 0 otherwise. If $Var(\Sigma') \setminus X$ is not empty, a variable v from this set is chosen, and the function `count_main` is recursively called on Σ' where v is assigned true and on Σ' where v is assigned false. The results returned by the two recursive calls are then summed up and multiplied by the variable *cpt* (line 14). Before returning the accumulated model count in *cpt* (line 17), the formula Σ is added to the cache associated with the corresponding projected model count *cpt* (line 15).

To integrate BCE into the search process (blue part), we first call the function `propagate` on S and *units* to update the information managed by the *bce* object and compute the set of blocked clauses B (line 3). Then, at line 4, we eliminate the stored set of blocked clauses B. To maintain consistency between the BCE manager information and the ongoing recursive call, the backtrack function must be executed before returning the calculated model count (lines 6 and 16).

4 Experimental Evaluation

Our aim was to empirically assess the advantages of employing blocked clause elimination in solving instances of the projected model counting problem. For our experimentation, we used 500 CNF instances from the three recent model counting competitions (the 2021, 2022, and 2023 editions documented at `https://mccompetition.org/`). We excluded instances from the 2020 competition due to incompatibility with our software caused by changes in

Algorithm 7 `count_main`.

```
Input: $\exists X.\Sigma$ an existentially quantified CNF formula.
Output: the number of models of $\exists X.\Sigma$ over $Var(\Sigma) \setminus X$

1  $(units, S, \Sigma) \leftarrow$ bcp($\Sigma$)
2  if $\perp \in \Sigma$ then return 0
3  $B \leftarrow bce$.propagate($S, \{x | \ell \in units$ and $Var(\ell) = x\}$)
4  $\Sigma \leftarrow \Sigma \setminus \{\Sigma[i]$ s.t. $i \in B\}$
5  if cache($\Sigma$) $\neq nil$ then
6      $bce$.backtrack()
7      return cache($\Sigma$)
8  $comps \leftarrow$ connectedComponents($\Sigma$)
9  $cpt \leftarrow 1$
10 for $\Sigma' \in comps$ do
11     if $Var(\Sigma') \setminus X = \emptyset$ then $cpt \leftarrow cpt \times$ (SAT($\Sigma'$)?1 : 0)
12     else
13         Let $v \in Var(\Sigma') \setminus X$
14         $cpt \leftarrow cpt \times$ (count_main($\Sigma' \wedge v$) + count_main($\Sigma' \wedge \neg v$))
15 cache($\Sigma$) $\leftarrow cpt$
16 $bce$.backtrack()
17 return $cpt$
```

the input format. The instances were categorized into three datasets: 200 from the 2021 competition, 200 from the 2022 competition, and 100 from the 2023 competition. Notably, as the full set of 2023 instances was unavailable at the time of writing, we only included the 100 public instances provided by the organizers.

The projected model counter used for the evaluation was `d4` [24]. Our experiments were conducted on Intel Xeon E5-2643 processors running at 3.30 GHz with 32 GiB of RAM, operating on Linux CentOS. Regarding the model counting competition, each instance was subject to a time-out of 3600 seconds and a memory limit of 32 GiB. For each instance, we measured the computation times required by three different versions of `d4` for counting the numbers of projected models. These versions include:

- `d4`: This is the standard version of `d4`, as given at `https://github.com/crillab/d4v2`.
- `d4+BCE`$_p$: This version of `d4` incorporates blocked clause elimination performed once during a preprocessing phase.
- `d4+BCE`$_i$: In this version of `d4`, blocked clause elimination is performed dynamically throughout the search achieved by the model counter.

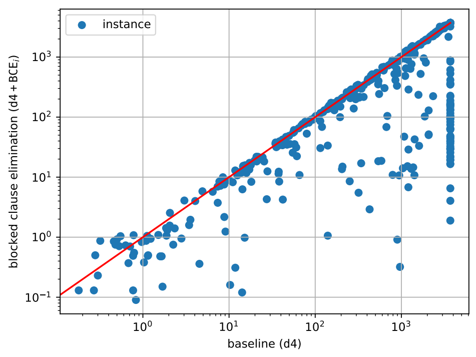
For all the versions under consideration, a preprocessing step of 60 seconds was conducted. This preprocessing involves running `BiPe` [27], followed by the occurrence elimination and vivification preprocessing for 10 iterations as described in [23] (we only replace the gate simplification with `BiPe`).

Table 1 presents the number of instances for which different versions of `d4` terminated within the specified time and memory constraints. The correctness of the extended versions of `d4` was verified by comparing their returned model counts with those of the baseline version. For all instances solved by the baseline version, the extended versions returned the same model counts. The table clearly demonstrates that leveraging dynamical blocked

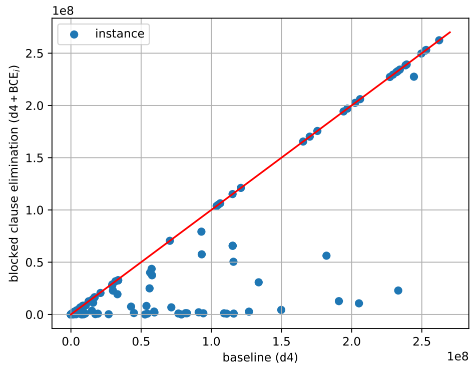
Table 1 The table shows the numbers of instances solved by different versions `d4` within a time limit of 3600 seconds and a memory limit of 32 GiB. The number of memory out (MO) are reported between brackets.

	2021 (200)	2022 (200)	2023 (100)	All (500)
`d4`	139 (56 MO)	149 (24 MO)	73 (9 MO)	361 (89 MO)
`d4+BCE`$_p$	139 (56 MO)	149 (25 MO)	73 (9 MO)	361 (90 MO)
`d4+BCE`$_i$	**172** (23 MO)	**163** (8 MO)	**78** (4 MO)	**413** (35 MO)

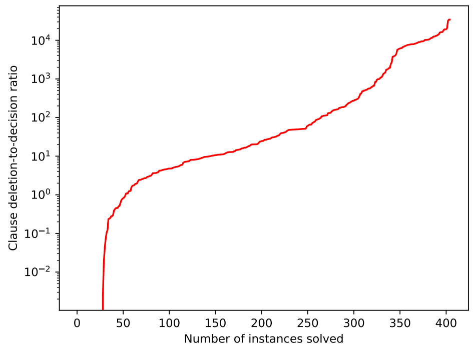
clause elimination significantly improves the performance of the model counter in practice. Furthermore, regardless the benchmark set considered, the version of `d4` equipped with dynamic blocked clause elimination systematically solved more instances than the two other versions. This indicates that the improvement is not limited to specific benchmark sets. Moreover, Table 1 shows that using blocked clause elimination solely during preprocessing phase did not lead to increase the number of instances solved. This demonstrates that for effective results, blocked clause elimination needs to be performed eagerly.



(a) Comparing the run times of `d4` and `d4+BCE`$_i$.



(b) Comparing the number of decisions made by `d4` and `d4+BCE`$_i$ on the instances solved by both.



(c) Plot used to estimate the extent of clause deletion relative to the number of decisions made for instances solved by `d4+BCE`$_i$.

Figure 1 Experimental results.

Figure 1a presents a pairwise comparison between `d4` and `d4+BCE`$_i$ on a scatter plot. Each data point represents an instance, with the x-axis indicating the time (in seconds) required to solve it using the baseline version of `d4`, and the y-axis representing the time for the enhanced version d4+BCE$_i$. The experimental results unequivocally demonstrate that the version of d4 with dynamic blocked clause elimination generally outperforms the baseline version of d4. Furthermore, the figure reveals instances where d4+BCE$_i$ achieves speeds one order of magnitude faster than the baseline version d4.

Focusing on instances solved by both approaches and exhibiting a solving time difference of more than five seconds, the baseline version d4 beat d4+BCE$_i$ for 75 instances, achieving an average speedup of 4%. This speedup is calculated as the ratio between the running times of the methods, with a peak improvement of 14% and the third quartile indicating a speedup of 7%. We investigated the factors contributing to the greater efficiency of the baseline version compared to the one employing dynamic blocked clause elimination, but we were unable to identify a definitive reason. It is hypothesized that the removal of clauses may negatively impact branching, potentially leading to a slightly larger search space explored by the model counter. On the contrary, we discovered 150 instances for which d4+BCE$_i$ outperformed the baseline version of d4. For them, d4+BCE$_i$ exhibits an average speed increase of 40 times compared to the baseline, with a peak improvement of up to 3000 times. The second quartile of the time distribution demonstrates a 50% improvement, while the third quartile shows a remarkable 600% enhancement.

Figure 1b showcases the number of decisions made by d4 and d4+BCE$_i$ on instances solved by both methods. This visualization sheds light on the fact that the performance enhancement cannot be solely attributed to a reduction in memory usage, which might otherwise account for the observed decrease in memory consumption with d4+BCE$_i$. It is widely recognized that the cache structure frequently serves as the primary memory bottleneck. Consequently, removing clauses reduces the size of cache entries, which typically results in decreased memory consumption and associated memory overhead. Nevertheless, as depicted in Figure 1b, the use of blocked clause dynamic elimination also results in a decrease in the number of decisions required by the model counter to complete its task. This underscores that the performance gain is not solely a consequence of an inadequate memory limit setting. Thus, even with a significant increase in the memory limit, employing dynamic blocked clause elimination proves highly advantageous in practice. Specifically, for 71 instances, d4 required fewer decisions than d4+BCE$_i$, with an average difference of 6460 decisions in favor of d4. The second quartile exhibited a difference of 102 decisions, while the third quartile showed a difference of 306 decisions across these instances. Conversely, d4+BCE$_i$ required fewer decisions than d4 for 231 instances, with an average difference of 12,249,678 decisions in favor of d4+BCE$_i$. The second quartile exhibited a difference of 33,005 decisions, while the third quartile showed a difference of 1,602,452 decisions across these instances.

Figure 1c gives the proportion of clauses removed through dynamic blocked clause relative to the number of decisions made for instances solved by d4+BCE$_i$. As observed in the plot, for approximately 300 instances, the average number of blocked clauses removed at each decision is at least 10. For about 100 of these instances, the average number of blocked clauses removed at each decision is at least 100. Additionally, for certain benchmarks, more than 1000 clauses where removed at each decision. While there is some variation in the extent of deletion across different steps, the plot clearly demonstrates that a substantial number of clauses are generally eliminated when employing dynamic blocked clause elimination.

5 Conclusion and Perspectives

In conclusion, this paper has explored the utilization of the blocked clause elimination dynamically during projected model counting. Despite its widespread application in the satisfiability problem, the blocked clause elimination rule posed challenges for model counting due to its inability to maintain the number of models unchanged. However, through focused attention on projected variables during the search for blocked clauses, we have demonstrated the feasibility of leveraging this rule while preserving the correct model count. To achieve this, we introduced a new data structure and corresponding algorithms tailored for leveraging blocked clause elimination dynamically during search. This innovative machinery has been integrated into the projected model counter `d4`, enabling us to conduct comprehensive experiments that showcase the computational benefits of our approach. Our results underscore the efficacy of leveraging the blocked clause elimination rule technique for projected model counting, opening avenues for further exploration and refinement in this domain.

Exploring extensions of blocked clause elimination (BCE) in the context of projected model counting is interesting future work. This particularly includes considering the elimination of resolution asymmetric tautologies (RAT) [17], or even covered [11, 5] or propagation redundant (PR) [13] clauses. These approaches hold the potential to uncover additional redundant clauses, that can be eliminated and thus improve efficiency of projected model counting. In addition, we envision the development of novel branching heuristics designed to prioritize the elimination of clauses that prevent removal of blocked clauses. These improved decision heuristics, could create more instances where clauses become blocked and thus eliminated, again with the goal to improve solver efficiency. Furthermore, we want to explore the applicability of blocked clause elimination to other reasoning tasks, particularly to the weighted `Max#SAT` problem [3, 2] or counting tree models of QBF formulas [21].

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

2 Gilles Audemard, Jean-Marie Lagniez, and Marie Miceli. A new exact solver for (weighted) max#sat. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 28:1–28:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.SAT.2022.28`.

3 Gilles Audemard, Jean-Marie Lagniez, Marie Miceli, and Olivier Roussel. Identifying soft cores in propositional formulæ. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 14th International Conference on Agents and Artificial Intelligence, ICAART 2022, Volume 2, Online Streaming, February 3-5, 2022*, pages 486–495. SCITEPRESS, 2022. `doi:10.5220/0010892700003116`.

4 Rehan Abdul Aziz, Geoffrey Chu, Christian J. Muise, and Peter J. Stuckey. #∃sat: Projected model counting. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 121–137. Springer, 2015. `doi:10.1007/978-3-319-24318-4_10`.

5 Lee A. Barnett, David M. Cerna, and Armin Biere. Covered clauses are not propagation redundant. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2020. `doi:10.1007/978-3-030-51074-9_3`.

6 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. `doi:10.3233/FAIA200992`.

7 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. 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 1468–1476. AAAI Press, 2020. doi:10.1609/AAAI.V34I02.5505.

8 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. 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 211–230. Springer, 2020. doi:10.1007/978-3-030-58475-7_13.

9 Johannes Klaus Fichte, Markus Hecher, Michael Morak, Patrick Thier, and Stefan Woltran. Solving projected model counting by utilizing treewidth and its limits. *Artif. Intell.*, 314:103810, 2023. doi:10.1016/J.ARTINT.2022.103810.

10 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020. doi:10.1007/978-3-030-51825-7_25.

11 Marijn Heule, Matti Järvisalo, and Armin Biere. Covered clause elimination. In Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, volume 13 of *EPiC Series in Computing*, pages 41–46. EasyChair, 2010. doi:10.29007/CL8S.

12 Marijn Heule, Matti Järvisalo, Florian Lonsing, Martina Seidl, and Armin Biere. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res.*, 53:127–168, 2015. doi:10.1613/JAIR.4694.

13 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. doi:10.1007/S10817-019-09516-0.

14 Marijn J. H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 183–212. IOS Press, 2021. doi:10.3233/FAIA200988.

15 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. doi:10.1007/978-3-642-12002-2_10.

16 Matti Järvisalo, Armin Biere, and Marijn Heule. Simulating circuit-level simplifications on CNF. *J. Autom. Reason.*, 49(4):583–619, 2012. doi:10.1007/S10817-011-9239-9.

17 Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning – 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi:10.1007/978-3-642-31365-3_28.

18 Benjamin Kiesl, Martin Suda, Martina Seidl, Hans Tompits, and Armin Biere. Blocked clauses in first-order logic. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 31–48. EasyChair, 2017. doi:10.29007/C3WQ.

19 Vladimir Klebanov, Norbert Manthey, and Christian J. Muise. Sat-based analysis and quantification of information flow in programs. In Kaustubh R. Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems – 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8054 of *Lecture Notes in Computer Science*, pages 177–192. Springer, 2013. doi:10.1007/978-3-642-40196-1_16.

20 Oliver Kullmann. On a generalization of extended resolution. *Discret. Appl. Math.*, 96-97:149–176, 1999. doi:10.1016/S0166-218X(99)00037-2.

21 Jean-Marie Lagniez, Florent Capelli, Andreas Plank, and Martina Seidl. A top-down tree model counter for quantified boolean formulas. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, 3rd-9th August 2024, Jeju, South Korea*. ijcai.org, 2024.

22 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artif. Intell.*, 281:103229, 2020. doi:10.1016/J.ARTINT.2019.103229.

23 Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pages 2688–2694. AAAI Press, 2014. doi:10.1609/AAAI.V28I1.9116.

24 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. doi:10.24963/IJCAI.2017/93.

25 Jean-Marie Lagniez and Pierre Marquis. On preprocessing techniques and their impact on propositional model counting. *J. Autom. Reason.*, 58(4):413–481, 2017. doi:10.1007/S10817-016-9370-8.

26 Jean-Marie Lagniez and Pierre Marquis. A recursive algorithm for projected model counting. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 – February 1, 2019*, pages 1536–1543. AAAI Press, 2019. doi:10.1609/AAAI.V33I01.33011536.

27 Jean-Marie Lagniez and Pierre Marquis. Boosting definability bipartition computation using SAT witnesses. In Sarah Alice Gaggl, Maria Vanina Martinez, and Magdalena Ortiz, editors, *Logics in Artificial Intelligence – 18th European Conference, JELIA 2023, Dresden, Germany, September 20-22, 2023, Proceedings*, volume 14281 of *Lecture Notes in Computer Science*, pages 697–711. Springer, 2023. doi:10.1007/978-3-031-43619-2_47.

28 Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res.*, 18:391–443, 2003. doi:10.1613/JAIR.1113.

29 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 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. doi:10.1145/378239.379017.

30 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019. doi:10.24963/IJCAI.2019/163.

31 Ralf Wimmer, Karina Gitina, Jennifer Nist, Christoph Scholl, and Bernd Becker. Preprocessing for DQBF. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 173–190. Springer, 2015. doi:10.1007/978-3-319-24318-4_13.

Speeding up Pseudo-Boolean Propagation

Robert Nieuwenhuis
Barcelogic.com, Barcelona, Spain

Albert Oliveras
Technical University of Catalonia, Barcelona, Spain

Enric Rodríguez-Carbonell
Technical University of Catalonia, Barcelona, Spain

Rui Zhao
Technical University of Catalonia, Barcelona, Spain

Abstract

Unit propagation is known to be one of the most time-consuming procedures inside CDCL-based SAT solvers. Not surprisingly, it has been studied in depth and the two-watched-literal scheme, enhanced with implementation details boosting its performance, has emerged as the dominant method.

In pseudo-Boolean solvers, the importance of unit propagation is similar, but no dominant method exists: counter propagation and watched-based extensions are efficient for different types of constraints, which has opened the door to hybrid methods. However, probably due to the higher complexity of implementing pseudo-Boolean solvers, research efforts have not focused much on concrete implementation details for unit propagation but rather on higher-level aspects of other procedures, such as conflict analysis.

In this paper, we present (i) a novel methodology to precisely assess the performance of propagation mechanisms, (ii) an evaluation of implementation variants of the propagation methods present in RoundingSat and (iii) a detailed analysis showing that hybrid methods outperform the ones based on a single technique. Our final contribution is to show that a carefully implemented hybrid propagation method is considerably faster than the preferred propagation mechanism in RoundingSat, and that this improvement leads to a better overall performance of the solver.

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, Pseudo-Boolean Solving, Implementation-level Details

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.22

Funding All authors are supported by grant PID2021-122830OB-C43, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF: A way of making Europe". Second and third authors are supported by Barcelogic through research grants C-11423 and C-11422, respectively.

1 Introduction

CDCL-based SAT solvers [25] have become the method of choice to solve a variety of problems coming from diverse areas such as system verification [10], security [17, 32], cryptography [31] and even mathematics [22]. Nevertheless, theoretical results [4, 27] have identified problems for which no polynomial CDCL execution exists. The reason for this is that resolution, the proof system on which SAT solvers are based, does not provide polynomial proofs for them [21]. This is particularly troublesome for problems, like the pigeon hole principle, which frequently appear in real-world problems [2]. Another well-known limitation of SAT regards the poor expressivity of its input language, which does not allow, for example, encoding numerical constraints in a natural way.

Pseudo-Boolean (PB) solving, also known as 0-1 Integer Linear Programming, has emerged as a remarkable alternative to SAT. The input language to PB solvers, consisting of 0-1 linear constraints, is an extension of propositional clauses, and allows for more compact encodings. In addition, CDCL-based PB solvers [28] are, at least from the theoretical point of view, exponentially more powerful than SAT solvers because they can reason using the cutting-planes proof system [13], which is exponentially stronger than resolution.

CDCL-based PB solvers incorporate many of the techniques that turned out to be essential in SAT solvers. Among them, a crucial procedure is unit propagation, which, given a clause where all literals are false except one that is unassigned, extends the assignment by adding this literal. The efficiency of this procedure is paramount for the overall performance of solvers, and hence efficient ways to implement it have been developed. Out of all these methods, the two-watched literal scheme [26] has established itself as the dominant method in SAT. It is based on a simple fact: if two non-false literals exist in a clause, no unit propagation is possible. Hence, two non-false literals are watched, and only when one of them becomes false, the clause is checked for propagation.

This idea is not so simple in PB constraints because the number of literals to be watched depends on its coefficients. Due to this reason, PB watched-literal propagation and counter propagation, the simpler version where all literals in every constraint are watched, have co-existed in PB solvers with no strong evidence of one method clearly outperforming the other: the PBS solver [3] used counter-based propagation without mentioning the possibility of using watches. The Galena [11] authors mentioned that watches are effective when the sum of the coefficients is large compared with the independent term. However, they opted to always use counters. Initially, the Pueblo solver [29] used counters but later moved to a version with watch propagation [30]. In the Sat4j [5] solver, both procedures are implemented and, according to the authors, not much difference was observed among them, with a slight preference for watches. Finally, RoundingSat [14] concluded that watched propagation was more efficient but, when specialized procedures for clauses and cardinality constraints are implemented, which is a standard technique used in all solvers, the difference is fairly small.

Surprisingly, with the remarkable exception of RoundingSat, it is not possible to find in the literature rigorous experimental evaluations that support any of these design decisions. The situation is dramatically different when it comes to SAT-based unit propagation. As an example, the cache behavior of watched-literal propagation has been thoroughly studied in a number of papers [33, 12, 23, 24], where important propagation speedups are reported. This caused solvers like MiniSAT [18], PicoSAT [6] or Lingeling [7] to incorporate carefully designed implementations of watched propagation [8]. A very pedagogical reference is the source code of CaDiCaL [9], where design decisions are discussed with pointers to the papers on which they rely.

This type of research efforts are very delicate because minimal changes on the implementation of propagation procedures can cause solvers to have dramatically different runtimes that are not directly related with the propagation scheme. This makes improvements that are not groundbreaking impossible to be perceived, because they are mostly blurred by the chaotic behavior of the solver.

In this paper we leverage SAT-solver implementers' knowledge and use that to improve the performance of PB solvers. For that purpose, we considered RoundingSat and improved the implementation of their propagation routines. We chose this solver because it is the most successful and actively maintained PB solver: its team has recently devoted considerable efforts to develop smart conflict analysis procedures [19], add core-guided techniques [16] and integrate MIP solvers [15], which has led RoundingSat to be "the world's fastest pseudo-Boolean solver" [1]. This speed is particularly striking because, in our opinion, its authors have focused more on the development of techniques that can give exponential improvements than on lower-level implementation aspects.

In order to evaluate the impact of the improvements we introduce, we also present a novel methodology that, by generating log files with the information of a solver execution, allows us to guarantee that the solver always traverses the search space in the same way. With this certainty, we can assess the precise impact of different unit propagation variants.

This paper is organized as follows. After some preliminaries in Section 2, we explain our novel methodology to compare propagation procedures in Section 3. Sections 4 and 5 present our improvements on counter and watched-based propagation, respectively. Section 6 reports on the development of a hybrid method for propagation and we conclude in Section 7.

Contributions

(I) First, we present a novel methodology for experimentally evaluating different propagation schemes. This has allowed us to detect small improvements that otherwise would have been unnoticed. It is well known in the solver developer community that by joining the forces of many small improvements, significant speed-ups may be achieved.

(II) Second, we describe and evaluate improvements on the implementation of counter and watch-based propagation in RoundingSat, giving concrete reasons for their impact.

(III) Third, we report on the use of hybrid approaches, where the decision to use counter or watched propagation is done per constraint. Although this was already mentioned in Galena and RoundingSat, where it was said to be not superior to counters in the first solver, and left as future work in the second, here we show that a hybrid approach clearly outperforms the virtual best solver that always chooses the best option between using counters or watches for *all* constraints.

(IV) Finally, we show that improvements in unit propagation do not negatively affect other components of the solver. That is, the improved solver is indeed faster and solves more instances than the original one.

2 Preliminaries

Pseudo-Boolean Constraints. Let $\mathcal{X}$ be a set of propositional variables. A *literal* is either a variable (x) or the negation of one ($\overline{x}$). We will assume that $\overline{\overline{x}} = x$. A *(pseudo-Boolean* or *PB) constraint* is a 0-1 linear inequality $\sum_i c_i l_i \geq d$ where the l_i's are literals and, without loss of generality, the c_i's (*coefficients*) and d (*degree*) are positive integers. When all coefficients are 1 we say that the constraint is a *cardinality constraint* and if, in addition, d is also 1 we say that it is a *clause*. A *formula* is a set of constraints.

Satisfiability. An *assignment* ρ is a set of non-contradictory literals. It is *total* if for any $x \in \mathcal{X}$ either $x \in \rho$ or $\overline{x} \in \rho$, and *partial* otherwise. A literal l is *true* in ρ if $l \in \rho$, is *false* if $\overline{l} \in \rho$ and is *undefined* otherwise. Given a constraint C of the form $\sum_i c_i l_i \geq d$, an assignment ρ *satisfies* it if $\sum_{i:l_i \in \rho} c_i \geq d$, and *falsifies* it if no extension of ρ can satisfy it. If we define $slack(C, \rho) = (\sum_{i:\overline{l_i} \notin \rho} c_i) - d$, it can be seen that ρ falsifies C if and only if $slack(C, \rho) < 0$. Note the slack expression sums the coefficients of all non-false literals, and that is the maximum value that the left hand side of the constraint can reach. If even that does not exceed d, no extension of ρ will do. An assignment that satisfies all constraints of a formula is called a *model*. Sometimes, instead of looking for any model, we are interested in finding one that minimizes a certain linear objective function $\sum_i a_i l_i$.

Unit Propagation. Given a constraint $C = \sum_i c_i l_i \geq d$ and an assignment ρ, we say that C *unit propagates* l_i under ρ if l_i is undefined in ρ, but l_i is true in any total assignment extending ρ that satisfies C. The latter is equivalent to checking whether $slack(C, \rho) < c_i$, i.e., if we do not set l_i to true, the constraint becomes falsified. Given a formula F and an assignment ρ, *unit propagation* of F under ρ is the outcome of applying the following two rules until a fixpoint is reached: (i) if ρ falsifies a constraint $C \in F$, a *conflict* is found with *conflicting constraint* C and we stop, (ii) if ρ unit propagates some literal l due to constraint C, extend $\rho := \rho \cup \{l\}$ with *reason* C.

Conflict-Driven Pseudo-Boolean Solving. A generalization of the well-known CDCL [25] algorithm for SAT can be applied to the pseudo-Boolean case [28]. The algorithm starts with an empty assignment ρ and proceeds as follows: (1) Apply unit propagation, possibly extending ρ. (2) If a conflict is found, a conflict analysis procedure derives a constraint C (called *lemma*) that can be safely added to the formula. If C is the constraint $0 \geq 1$, the formula is unsatisfiable, otherwise it is guaranteed that after removing some literals from ρ in a last-in first-out way (*backjumping*), C allows some literal to be unit propagated. Hence, we go to step 1. (3) If no conflict is found, and ρ is total, it is a model of the formula. Otherwise, an undefined literal l (*decision literal*) is added to ρ and we go to step 1. The choice of l is determined by sophisticated heuristics.

Since the set of constraints might grow too much, periodically a *cleanup* process that deletes some lemmas is performed. Also periodically, the process *restarts*: roughly speaking, ρ is reset to the empty set. Since all lemmas in the formula are still kept, the behavior of the algorithm will change.

3 Design of a Fair Evaluation of Different Propagation Mechanisms

Developing a state-of-the-art PB/SAT solver not only consists in implementing and combining the appropriate techniques. Optimizing its implementation is a critical task, which turns out to be very complex due to the "chaotic" behavior of these solvers. On a single instance, seemingly innocuous changes may have a strong impact on the overall solver runtime, but this does not necessarily mean that this change is worth incorporating to the solver. One prominent example of this phenomenon takes place when optimizing unit propagation. A simple change in the order in which clauses are processed during this procedure drives the solver towards finding one conflicting clause or another, which in turn, leads to different learned lemmas and to completely different search space traversal. Hence, evaluating the impact of a new implementation technique on a small set of benchmarks is hopeless. This is usually fixed by performing exhaustive experiments on very large sets of benchmarks, hoping that, by evaluating some measure over all the collected data, one will be able to conclude whether this implementation method pays off or not. In our opinion, this has at least two problems. Firstly, it makes the evaluation of these improvements a very time-consuming task. Secondly, modest improvements are not observable with this methodology, because the noise produced by the changes in search space exploration end up hiding their real impact.

In order to avoid this troublesome scenario, we want to force the solver to explore the search space in the same way, independently of the unit propagation mechanism we implement. The good news is that the outcome of the unit propagation procedure is almost unique. If there is no conflict, any complete propagation procedure will add exactly the same literals to the assignment. The only difference is that they may be added in a different order, or due to different reasons, but we will see that this is not a problem for our method. Otherwise, if unit propagation derives a conflict, any complete procedure will find it. However, since propagation stops as soon as it finds the first conflicting clause, different propagation variants may find different conflicting clauses, which will lead to the generation of different lemmas by conflict analysis and subsequent different search space exploration.

Intuitively our methodology relies on an *oracle* that, after a conflicting clause is found, tells the solver which lemma should be learned. In principle, that would allow the solver to always explore the search space in the same way[1]. However, there are other ingredients in the CDCL procedure that have a strong effect on the search behavior: decision heuristics, cleanups and restarts. For that reason, our oracle will in addition have to inform the solver about (i) which literal should be the next decision, (ii) when the next cleanup should be applied and which constraints should be removed, and (iii) at which point a restart should be applied.

In this paper, the role of the oracle will be played by a *log*, which is a file obtained from the execution of a PB solver where all the previously mentioned information is written. It is not difficult to modify a PB solver so that it reads these logs. The solver then mostly consists of only the unit propagation procedure, since the outcome of all other computationally-expensive tasks is provided by the oracle. With this infrastructure setup, we can change the unit propagation procedure with the guarantee that the search behavior of the solver (i.e. number of decisions, number of conflicts, etc.) will be the same.

Since RoundingSat[2] is probably the fastest existing CDCL-based PB solver, we have chosen it in order to conduct our work. In the rest of the paper, except in the very last experiment, RoundingSat is always executed reading the logs we previously generated. This experimentation[3] was done on 3.3Ghz 16GB Intel Xeon E-2124 machines over a small (about 100) set of benchmarks we selected from the OPT-SMALLINT-LIN (optimization problems with small integers and linear constraints) category of the 2016 Pseudo-Boolean Competition[4], the last edition that took place. Benchmarks were chosen so as to contain a variety of runtimes, ranging from easier (a few seconds) to more difficult ones (more than one hour).

4 Counter-Based Propagation

RoundingSat follows the standard way to implement propagation in Conflict-Driven Learning solvers: for each literal l, a vector $wlist(l)$ (the *watch list* of l) contains a superset of the constraints[5] that might have become false or propagating due to adding $\bar{l}$ to the assignment. Hence, even by visiting only these constraints, all propagations and conflicts are detected. Let us see in detail how this is implemented in RoundingSat.

In counter-based propagation, every constraint C of the form $\sum_i c_i l_i \geq d$ has an element of type *Watch* in $wlist(l_i)$ for all i. This element is a pair $\langle ctrPtr, idx\rangle$, where $ctrPtr$ can be seen as a pointer to C and idx is the position of l_i in C. At some point after a literal l is added to the assignment, l is *checked for propagation*: all elements of $wlist(\bar{l})$ are traversed and the function in Algorithm 1 is called, which assumes that constraints are always sorted from largest coefficient to smallest.

[1] In conflicting decision levels, the number of propagated literals may still be different due to the possibility of finding different conflicting clauses. Fortunately, this had a negligible impact on propagation time.
[2] In order to reproduce the results in [14] we have worked on the version used in that paper. However, we want to remark that the newest version of RoundingSat has not changed its propagation routines.
[3] Additional material can be found in `https://github.com/dearzhaorui/speedup-roundingSat`.
[4] `https://www.cril.univ-artois.fr/PB16/`
[5] Note that watch lists also contain clauses and cardinality constraints, which have specialized propagation procedures, but we mostly ignore them in this paper. We added binary clauses as another particular type of constraint, but this did not change the performance significantly.

Algorithm 1 Counter-based propagation procedure.

```
1  Function Propagate-Counter( $Watch\ w$ ):
2      Constraint $ctr := w.ctrPtr$
3      if $isDeleted(ctr)$ then return
4      if $not\ isPBCounter(ctr)$ then return
5      $coef := ctr[w.idx].coef$
6      $maxCoef := ctr[0].coef$
7      $ctr.slack := ctr.slack - coef$
8      if $ctr.slack < 0$ then return $CONFLICT$
9      if $ctr.slack < maxCoef$ then // possible propagation
10         $i := 0$
11         while $i < ctr.size$ and $ctr.slack < ctr[i].coef$ do
12             if $isUndef(ctr[i].lit)$ then $propagate(ctr[i].lit)$
13             $i := i + 1$
14     return $OK$
```

Let us first remark that lines 3 and 4 are not part of a standard counter propagation routine. However, for every element in the watch list, RoundingSat first checks whether it is a deleted constraint, and then checks the type of the constraint in order to decide which propagation mechanism is invoked. We added them to Algorithm 1 to make it evident that in RoundingSat these two lines are always executed.

If we consider $\hat{\rho}$ to be the set of literals in the current assignment ρ that have been checked for propagation, the *slack* field in a constraint stores the sum of the coefficients of the non-false literals w.r.t $\hat{\rho}$ minus the degree, i.e. $slack(C, \hat{\rho})$. Since $\hat{\rho} \subseteq \rho$, it holds that $slack(C, \rho) \leq slack(C, \hat{\rho})$. If this is a negative value (line 8), the constraint is falsified by $\hat{\rho}$ and hence by ρ.

In order to check for propagation, if $maxCoef$ is the maximum coefficient in the constraint we know that if $slack(C, \hat{\rho}) \geq maxCoef$, then $slack(C, \hat{\rho}) \geq c_i$ for all i and hence $\hat{\rho}$ does not propagate any literal. This is checked in line 9. Otherwise, the constraint is checked for propagation w.r.t $\hat{\rho}$. Note that since the constraint is ordered by coefficient, once a literal is not large enough to be propagated, none of the subsequent literals will be (second condition in line 11). A final remark is that eventually all literals are checked for propagation ($\hat{\rho} = \rho$) and hence all conflicts and propagations w.r.t. ρ are finally computed.

Algorithm 1 is in fact a simplified version of the one in RoundingSat [14], which contains an important optimization that allows the loop at line 11 to not always start with $i = 0$. More concretely, if one execution of the loop stops at some position $k < ctr.size$ and no backjump is applied, the next traversal of the constraint can start at position k. This improvement will be present in all our variations of counter-based propagation because we independently confirmed that it does indeed improve the performance.

Minimizing the Number of Constraint Loads. As we have mentioned, when a literal l becomes false, the watch list of $\bar{l}$ is traversed, checking constraints in that list for propagation. This has no remarkable difference with what happens in SAT solvers. Since constraints are consecutively accessed in the order in which they appear on the watch list, one would like to keep them as close in memory as possible, so as to minimize the number of cache faults. This is usually done with a class (ClauseAllocator in systems deriving from MiniSAT or Arena in CaDiCaL or Kissat [9]) that encapsulates the clause memory management. However, due to the huge number of clauses, this is still a bottleneck in SAT solvers. As the CaDiCaL source code literally says: "the cache line with the clause data is forced to be loaded here and thus this first memory access below is the real hot-spot of the solver".

Hence, in a PB solver, when we visit constraints we should try to avoid loading the constraint into memory as much as possible. This is something that, as we will see, was not considered in RoundingSat, where priority was given to developing an extremely modular and flexible system where all information is kept in their expected data type. This is a very natural decision design that facilitates extending the system with more sophisticated reasoning techniques, but has some impact on efficiency.

More concretely, a *Constraint* object in RoundingSat contains, among others, the following fields: a C-like array of 32-bit integers with its coefficients and literals, a 64-bit integer storing the degree, a 64-bit integer for the slack (where the 3 smallest possible values are reserved to express that this constraint is a clause, a cardinality constraint or a PB constraint), a 64-bit integer that uses 1 bit to express whether the constraint is deleted, 1 bit to indicate whether counter or watch propagation is to be used for this constraint, 30 bits for the size, and some others bits for information that is not relevant for our purposes.

A quick analysis of Algorithm 1 reveals that the solver always executes line 3, which forces the constraint to be loaded into memory, because the deletion information is inside the constraint. In order to minimize the number of times that the constraint is loaded (the hot-spot of the solver) we suggest to store all necessary information about a constraint in the watch list by modifying the structure *Watch*. More concretely:

- The type of constraint (binary clause, clause, cardinality constraint, counter PB constraint, watched PB constraint), needed in line 4 of Algorithm 1, will be stored in Watch.
- The coefficient of the watched literal (line 5) will also be stored in Watch.
- The slack of the constraint cannot be stored in Watch, because modifying it (line 7) would require traversing all other watch lists where this constraint occurs. Hence, we create a vector of 64-bit integers, indexed by a constraint identifier. This is much better from the memory point of view, because all slacks are in contiguous memory, unlike what happens with constraints.
- The maximum coefficient of the constraint will not be stored anywhere. What we suggest is that the slack vector contains an integer corresponding to the definition of $slack(C, \hat{\rho})$ *minus* the maximum coefficient. We call this vector $slackMM$ and will be enough to detect conflicts and propagations.
- The information about whether a constraint is deleted or not could be stored in Watch. However, that would make cleanup slightly slower since whenever a constraint is marked for deletion, the corresponding watch lists should be traversed. What we suggest is to keep this information in the $slackMM$ vector, where 1 bit corresponds to deletion information and the rest to the value we have described before.

Summing up, for PB constraints the Watch structure needs three pieces of information: the identifier of the constraint, the index of the literal in the constraint, as well as the coefficient[6]. The new structure will contain 3 integers in such a way that they also suffice to also store all the additional information we need (type of constraint, the other literal in binary clauses, a cached literal[7] for long clauses, and the index of the literal in a cardinality constraint):

- *internal_id:* stores the identifier of the constraint, only needed for PB constraints.

[6] The index is indeed not necessary for counter-based propagation. However, it will be needed for watch-based propagation. Since our goal is to develop a hybrid method, we store the index here as well.

[7] Sometimes called blocking literal, it is a well-known technique in SAT-based unit propagation that prevents loading the clause in some situations.

Algorithm 2 Improved counter-based propagation procedure.

```
1  Function Improved-Propagate-Counter(Watch w):
2      $id := w.identifier()$
3      if $slackMM[id].isDeleted$ then return
4      if $w.type() \neq PB\text{-}counter$ then return
5      $coef := w.coefficient()$
6      $slackMM[id].slack := slackMM[id].slack - coef$
7      if $slackMM[id].slack < 0$ then // possible prop., but check conflict
8          Constraint $ctr := constraints[id]$
9          $slack := slackMM[id].slack + ctr[0].coef$  // ctr sorted by coef.
10         if $slack < 0$ then return CONFLICT
11         $i := 0$
12         while $i < ctr.size$ and $slack < ctr[i].coef$ do
13             if $isUndef(ctr[i].lit)$ then $propagate(ctr[i].lit)$
14             $i := i + 1$
15     return $OK$
```

- *idx_type:* this is used for storing the index of the literal and partially, the constraint type. For binary clauses this integer is equal to -1; for longer clauses to an integer smaller than -1; for cardinality constraints, the last bit of this integer is 1, and the rest of the bits store the index of the literal in the constraint; finally, for PB constraints its last bit is 0, and the rest of the bits contain the index. Note that this yet does not allow us to distinguish between counter or watch-based constraints. This will be done using the sign of the next field.
- *coef_lit:* for bin clauses, it contains the other literal in the clause. For PB constraints, it contains the coefficient of the literal for counter-based constraints, which is always positive, and minus the coefficient for watch-based constraints. This field is not used for long clauses or cardinality constraints.

The Constraint structure is the same except for the slack information, that has been removed. All these changes allow us to implement the more efficient version of counter-based propagation of Algorithm 2.

Note that lines 3, 4 and 5, that check for deletion, constraint type and retrieve the coefficient, do not load the constraint yet and only require access to information in the Watch element. If line 6 is executed, $slackMM$ will be loaded into memory. Only if the check in line 7, that corresponds to line 9 in Algorithm 1, succeeds we load the constraint into memory in line 8. In this case, we recompute the actual slack of the constraint in line 9 and check for conflict afterwards. The rest of the algorithm is unchanged.

We computed, for every benchmark in our suite, the percentage of Watch elements of type PB for which we loaded the constraint, i.e, for which line 8 was executed and obtained an average of 6.29% and a median of 1.26%. This is a remarkably low number that shows that the number of constraint loads is reduced dramatically by Algorithm 2.

In addition, in [14] it is explained how to restore the slack information upon backjumping: whenever a literal is removed from ρ, the watch list of $\bar{l}$ is visited and for every PB constraint in that list, its slack is increased by the coefficient of l in the constraint. This again requires accessing all constraints, whereas in our implementation, only the $slackMM$ vector will be loaded into memory.

An analysis of the impact of this improvement can be seen in the top-left scatter plot of Figure 1. Plots of this type will always compare a baseline system with an enhancement, with a caption of the form "Enhancement vs. Baseline". A point $(100, 2)$ represents a benchmark for which the baseline version took 100 seconds and the enhancement took $100/2 = 50$ seconds, whereas a point $(100, -2)$ corresponds to a benchmark for which the baseline version took 100 seconds and the enhanced one took $100 \cdot 2 = 200$ seconds. Hence, points with positive y's always represent benchmarks for which the enhancement was indeed faster. Note the logarithmic scale on the x axis and that, for each plot, the scale on the y may be different. For the top-left scatter plot in Figure 1, the baseline system (*original counter*) corresponds to a variant of RoundingSat that uses counter-based propagation for all constraints except for cardinality constraints and clauses.

Revisiting Garbage Collection Frequency. In order to evaluate the impact of each of the improvements we present, we will always consider as a baseline the system that implements all previous modifications. Hence, we now consider as the reference a system where most information has been moved out of the Constraint class and we assess what is the gain obtained by adding the new improvement we present next.

Whenever a cleanup is performed, RoundingSat does the following: the set of constraints is traversed and some of them are marked as deleted, changing the bit allocated for this inside $slackMM$. In this case, a variable that contains the amount of memory that is occupied by deleted constraints is increased. If this variable exceeds a certain threshold, garbage collection is applied: all surviving constraints are reallocated contiguously in the database and watch lists are rebuilt with no reference to the deleted constraints.

In our opinion, this has two unwanted effects. First of all, watch lists are larger than strictly necessary, and this can slow down the system. Secondly, the constraints database is not as compact as possible and accesses to constraints will be more costly. We suggest to apply garbage collection in every cleanup. This would probably not pay off in a system that applies cleanups extremely often, because the extra cost of applying the garbage collection procedure at every cleanup would exceed the speedups obtained during propagation. Since after this change there will be no deleted constraints in any watch list, it is no longer necessary to use one bit of $slackMM$ to store this information. This will slightly speed up the access to the slack information, that will now occupy all 64 bits of $slackMM$.

As the middle scatter plot in the first row of Figure 1 shows, this modification has beneficial effects in almost all benchmarks. This is obviously a less remarkable improvement than the first one, but our novel methodology has allowed us to detect that this indeed enhances the runtime and, moreover, is extremely easy to implement.

Deletion of Elements in Watch Lists During Propagation. When performing propagation in a counter-based system, no PB constraint is ever deleted from any watch list. This is because all literals in every constraint are always "watched". However, let us remember that watch lists also contain clauses, for which the two-watched literal scheme is used. In this method, literals are sometimes unwatched, and this forces the corresponding element in the watch list to be deleted. The same phenomenon happens with watched-literal extensions for propagating cardinality constraints.

In order to use contiguous memory, watch lists in RoundingSat, and in state-of-the-art SAT solvers, are implemented as vectors. This makes it difficult to delete an element which is not the last one. In RoundingSat, this is done by copying the last element of the vector to the position of the element to be deleted, and then removing the last element. However, in SAT solvers like MiniSAT, CaDiCaL and Kissat, another solution is proposed.

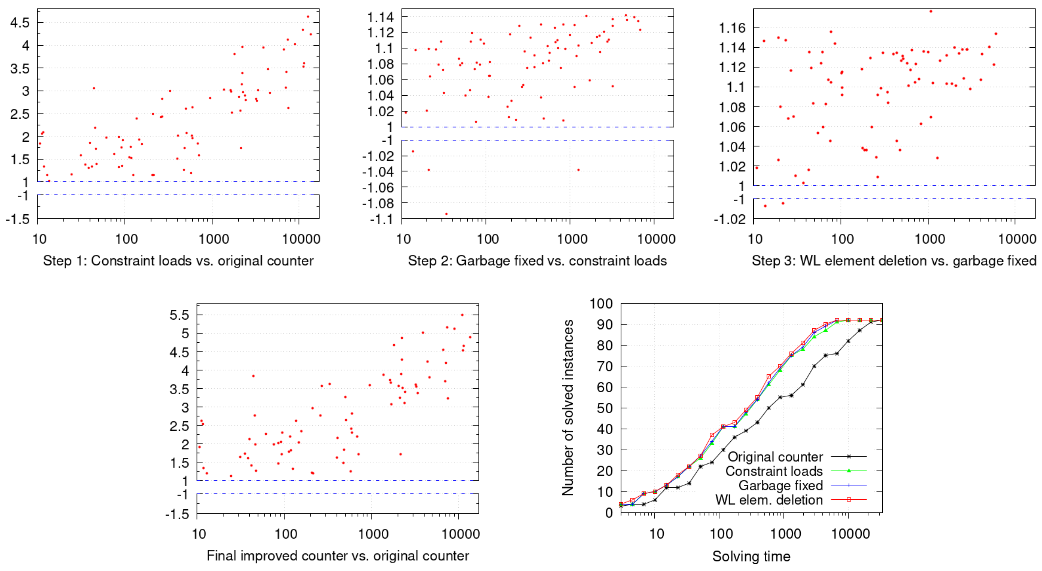


Figure 1 Impact of several improvements on counter-based propagation.

The idea is to read the watch list and, at the same time, write in the same vector the list that will result after the deletions have been performed. This is possible by keeping two pointers, one pointing to the position we are reading, and the other one to the position where we are writing. Since the first will always be to the right (or at the same position) of the second one, both things can be done at the same time without the two processes interfering with each other.

In terms of memory access this latter mechanism is better than the solution used in RoundingSat which, in the presence of long lists, often accesses positions in the vector that are very far away. This is what happens in counter-based propagation, where the fact that all literals are watched creates watch lists that are very long. On the other hand, one could argue that the solution used in SAT solvers incurs in some overhead due to performing a larger number of write operations. This, for example, would occur in a list where only the first element has to be deleted: RoundingSat would only perform one write operation whereas the other solution has as many writes as elements in the vector.

The rightmost plot in the first row of Figure 1 shows that implementing the SAT approach to deleting elements on top of the system that incorporates the two previous adjustments results in improvements in almost all benchmarks. Again, gains are limited. However, the addition of the last two small enhancements has some positive impact. This can be seen by comparing the leftmost scatter plot in the second row (a comparison between the original RoundingSat system and the one with all improvements) and the leftmost plot in the first row (a comparison between the original RoundingSat versus the system that only incorporates the first major improvement). Note that the distribution of the points is very similar, but the y scale is larger.

All this is summarized in the cactus plot of Figure 1. Note that there was no time limit in this experiment: we let all systems run until they processed the entire log. This is why the cactus plot shows that all systems end up processing all benchmarks. However, if a more strict time limit was given, the performance of the system with all improvements outperforms the system with only the first one, which is of course the major contributor to the overall speedup of the system.

Algorithm 3 Watch-based propagation procedure.

```
 1 Function Propagate-Watch(Watch $w$):
 2     Constraint $ctr := w.ctrPr$
 3     if $isDeleted(ctr)$ then return
 4     if not $isPBWatched(ctr)$ then return
 5     $coef := ctr[w.idx].coef$
 6     $maxCoef := ctr[0].coef$
 7     $ctr.wslack := ctr.wslack - coef$
 8     $i := 0$
 9     while $i < ctr.size$ and $ctr.wslack < maxCoef$ do
10         Lit $l := ctr[i].lit$
11         if not $isFalse(l)$ and not $ctr[i].watched$ then
12             $ctr[i].watched := true$
13             $wlist(l) := wlist(l) \cup \{\langle ctr, i \rangle\}$
14             $ctr.wslack := ctr.wslack + ctr[i].coef$
15         $i := i + 1$
16     if $ctr.wslack \geq maxCoef$ then
17         $ctr[w.idx].watched := false$
18         $wlist(ctr[w.idx].lit) := wlist(ctr[idx].lit) \setminus \{\langle ctr, w.idx \rangle\}$
19         return $OK$
20     if $ctr.wslack < 0$ then return $CONFLICT$
21     $j := 0$
22     while $j < ctr.size$ and $ctr.wslack < ctr[j].coef$ do
23         if $isUndef(ctr[j].lit)$ then $propagate(ctr[j].lit)$
24         $j := j + 1$
25     return $OK$
```

5 Watch-Based Propagation

Watch-based propagation can be seen as a refinement of counter-based propagation. The idea is to associate to each constraint a set of watched literals $watches(C)$. In RoundingSat, for each literal $l \in watches(C)$, an entry of type $Watch = \langle ctrPtr, idx \rangle$ is added to $wlist(l)$, and only when l becomes false we will traverse $wlist(C)$, and hence C, checking for a propagation or a conflict. In counter-based propagation, $watches(C)$ consists of all literals in C and hence, whenever any literal in C becomes false a visit to the constraint is triggered. In watch-based propagation, we will do so only when some literal in $watches(C)$ becomes false. Hence, a reduction in the size of $watches(C)$ will reduce the amount of work to be done.

Following the notation in [14], if we define $watchslack(C, \rho) = (\sum_{\substack{i:\overline{l_i} \notin \rho \\ l_i \in watches(C)}} c_i) - d$ we can easily see that $watchslack(C, \rho) \leq slack(C, \rho)$ and hence if $watchslack(C, \rho) \geq maxCoef(C)$ we can guarantee that C is neither conflicting nor propagating. Hence our goal is to watch a small set of literals for which the previous inequality holds. Note also that $watchslack(C, \rho)$ only changes if some $l \in watches(C)$ becomes false. All these remarks are the core of watch-based propagation routine in RoundingSat that we show in Algorithm 3.

It starts by decrementing the *watchslack* in line 7. If, after that, it is still larger than the maximum coefficient the algorithm skips the loop in lines 9–14 and executes lines 16–17: the literal that became false is removed from the *watches* set by marking it as unwatched in *ctr* and removing it from the corresponding watch list. Otherwise, the loop in lines 9–14 tries to extend the set *watches* so that the *watchslack* is larger than the maximum coefficient. If this is possible, lines 16–17 are executed. Otherwise, if the *watchslack* is negative, a conflict is declared in line 19. Finally, if *watchslack* is not larger than the maximum coefficient, but it is not negative, a propagation might be possible. This is checked in lines 20–23 by traversing the constraint and looking for literals to be propagated. For a more detailed description and theoretical justification, we refer the reader to [14].

Again, this is a slightly simplified version of what is done in RoundingSat. In particular, the same optimization that we mentioned in counter-based propagation that allows one to not always start the loop in lines 20–23 at position 0 is applied. Also, if the function is called on a certain constraint and the loop in lines 9–14 cannot extend *watches* so that *watchslack* is larger than the maximum coefficient, we know that in any subsequent call to this routine on the same constraint that is made with no backjump in between, we can skip this loop because it will not succeed either. These two optimizations are present in all our modifications of the algorithm.

Minimizing the Number of Constraint Loads. As we mentioned in the previous section, our aim is to reduce the number of times when a constraint is loaded into memory, because it is a very time-consuming task. If we analyze the code, we can see that, if after decreasing the *watchslack* it is still large enough, we have to unmark the coefficient, which is information stored in the constraint. In particular, RoundingSat uses the sign of the coefficient (which we know are always positive) to indicate whether the literal is watched or not. Otherwise, if we have to extend the set of *watches* we obviously have to load the constraint into memory as well. Hence, there does not seem to be too much room for reducing the number of constraint loads.

One could try to develop a sophisticated way to store outside of the constraint object the information about who is watched in every constraint. That would allow us to avoid loading the constraint when the *watchslack* is large enough even after we decrease it. Our experimental analysis showed that this situation happens in around 20% of the cases. Hence, we believe there is still some space for improvement by a major modification of the propagation scheme.

Despite our initial analysis of this improvement was not very optimistic, our results show that, by applying the same modification we described for counter-based propagation we still obtain propagation speed-ups, as the leftmost plot in the first row of Figure 2 shows. This is due to at least three reasons. First of all, for deleted constraints we do not have to load the constraint into memory. Secondly, checking the type of constraint that the $Watch$ element contains, in order to decide which propagation routine should be called, does not require loading the constraint. This is beneficial, for example, in constraints like clauses where the propagation mechanism does not require to always load the constraint into memory. Finally, backjumping traverses watch lists and updates the *watchslack* of the constraints appearing in them. Having moved this information outside of the constraint prevents the system from loading all these constraints into memory.

Revisiting Garbage Collection Frequency. Note that, as we did with counter-based propagation, we evaluate improvements by comparing them with a system that already incorporates all previously proposed adjustments. As it happened with counter-based propagation, incrementing the garbage collection frequency only produces some moderate improvements. The center plot in the first row of Figure 2 shows that the improvements are even more limited than with counter-based propagation. The most plausible reason is that, since deleted constraints only appear in the watch lists of its watched literals, which might be a very small subset of them, the number of elements marked for deletion in these lists is not large enough to be a problematic issue.

Deletion of Elements in Watch Lists During Propagation. It was unexpected to us that, as the rightmost plot in the first row of Figure 2 indicates, modifying the deletion of elements in watch lists to mimic the way it is done in SAT solvers slowed down the propagation procedure. However, a more careful analysis allowed us to realize that this should not have been such a big surprise.

The reason for this behavior is the length of the watch lists. As we mentioned, the original way of RoundingSat to delete elements in watch lists accesses the last element of the list, and this can be harmful due to memory access issues if the list is very long. For counters, the average watch list length over the benchmarks we are considering was 2365, and the median 564. For watch-based propagation, this figure was much lower: 203 on average and a median of 71. Hence, our concerns about having too long lists do not apply in watch-based propagation. However, since our goal is to develop a hybrid approach where some constraints will be watched and some others will be counter-based, we will keep this modification.

Circular Search for Watched Literals. In [20], a simple but effective improvement on the two-watched literal scheme for propagation in SAT solvers was described. The idea is that, whenever the clause is traversed in order to find another literal to watch, instead of starting from the beginning of the clause, as it was usually done, we search for it in a *circular* way. More precisely, the novel procedure stores with every clause the position in which the last search for new literals to be watched stopped. Next time the same operation is performed, the search starts from that position instead of from the beginning.

The reason why that is beneficial is that it is known that one prefers to watch literals that are inactive, i.e., that are rarely added to the assignment, because they cause almost no work in propagation. By searching in a circular way, all literals have the same chances to be watched and hence, if inactive literals exist, it will be more likely that the procedure ends up watching them.

As the leftmost plot in the second row of Figure 2 shows, this also pays off in the case of watch-based propagation for PB constraints. Note that, since we only wanted to evaluate the impact of this modification in the propagation of PB constraints, it has not been incorporated to the specialized propagation procedures for clauses or cardinality constraints.

A summary on the impact of all these improvements can be seen in the center scatter plot in the second row of Figure 2, where quite a consistent improvement can be observed. We want to remark that these are important speedups. We have to consider that we did not change the algorithmic nature of the propagation algorithms, but rather modified implementation details of them. Since we execute all systems on the same log, the search space is the same and we cannot expect exponential improvements in time.

Once we have developed our improved version of counter and watch-based propagation, it is worth comparing them and analyze whether one of them dominates the other. This is done in the last row of Figure 2 where, in the leftmost plot, we compare the performance of the two original methods in RoundingSat. It is clear that watch-based propagation outperformed counters. This was already mentioned in [14], but our more precise evaluation methodology allows us to conclude that the difference is probably larger than what it was reported in that paper. The analysis for the improved versions, in the rightmost plot of the same figure, reveals that they are quite similar. Since we now have two methods that are comparable, it makes even more sense to try to combine them in a hybrid procedure that determines which propagation method to use for each constraint.

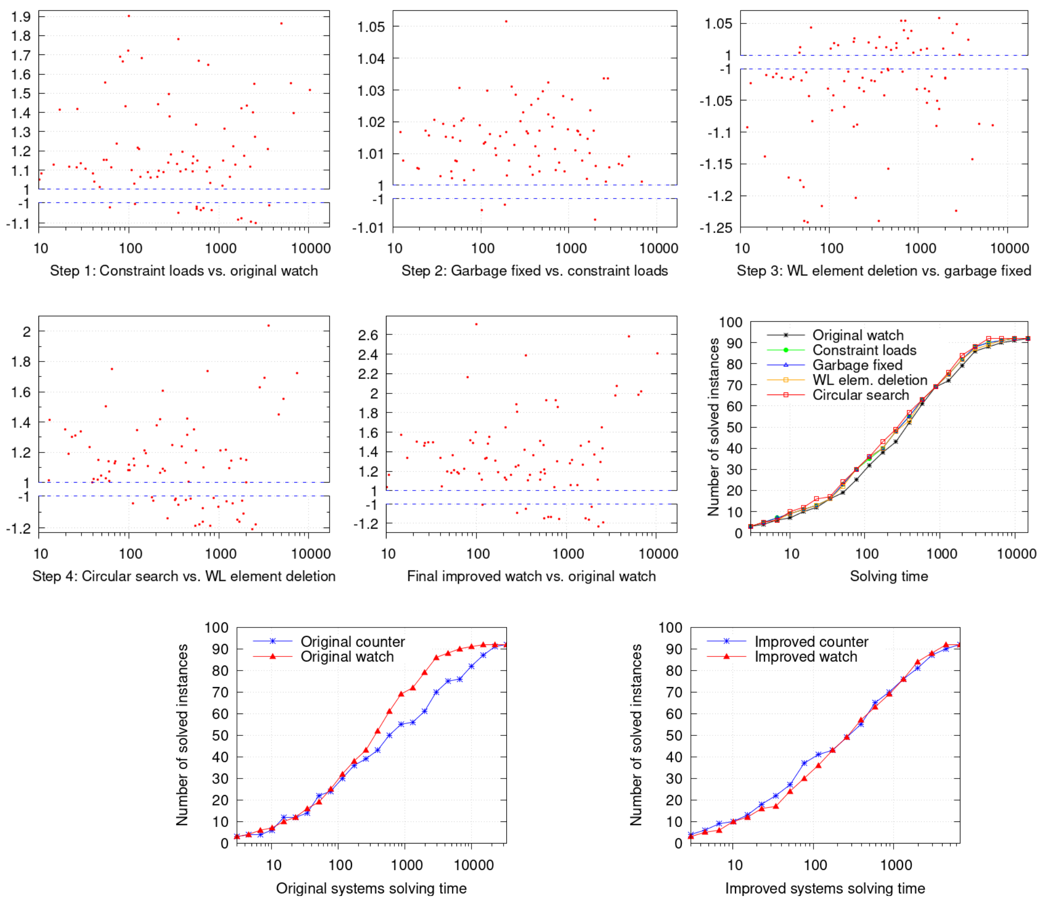


Figure 2 First two rows show the impact of several improvements on watch-based propagation. Last row compares, for the original and the improved versions, the performance of counter with respect to watched-based propagation.

6 A Hybrid Pseudo-Boolean Propagation Approach

The idea of using a *hybrid* approach where the propagation method is decided per constraint is not a new one. Already in [11] it was mentioned that “Another experiment assessed the performance of a hybrid BCP scheme. [...] The hybrid scheme was not found to be superior to the simple counter scheme.”. Unfortunately, the results of that experiment are not reported in the paper. In [14] this possibility is left as future work.

However, RoundingSat indeed incorporates a hybrid propagation mechanism: whenever a constraint is added to the database, it is first sorted from largest to smallest coefficient. After that, the smallest prefix of the constraint whose *watchslack* is larger than the maximum coefficient is computed. If the percentage of literals not in this prefix (the literals that will not be watched) is larger than a predefined threshold, watches are used. Hence, the larger the threshold, the larger the number of constraints for which counter-based propagation will be used. The rationale is that watches are preferred over counters when a sufficiently small percentage of the literals are watched.

The default version in RoundingSat uses watch-based propagation. However, one can instruct the solver to use the hybrid propagation mechanism, but a concrete percentage threshold has to be given. An extensive evaluation allowed us to conclude that 0.8 was the best threshold when using this hybrid method on top of the original RoundingSat

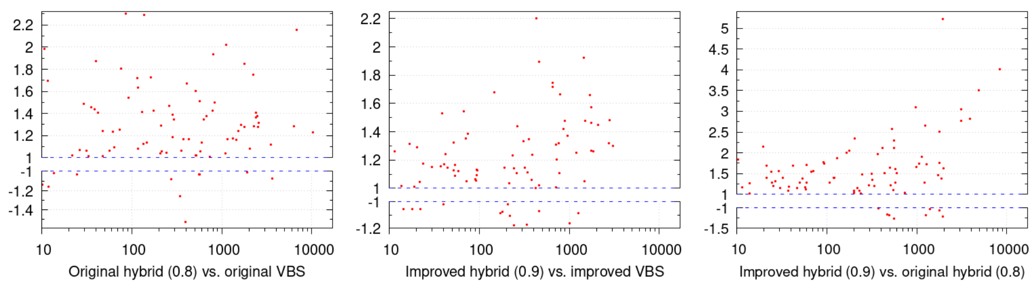


Figure 3 Comparison between different hybrid methods. Leftmost plot uses the original counter and watch-based propagation routines, whereas the center plot uses the improved ones. Rightmost plot compares a hybrid method on top of them.

propagation routines. The leftmost plot in Figure 3 compares this method with a Virtual Best Solver (VBS) that, for each benchmark, selects the best possibility between using counters or watches for all constraints. When using the hybrid approach on top of our improved propagation routines, 0.9 was the best threshold and a comparison with the improved VBS is in the center plot. We want to remark that improving upon a VBS is always a challenging task.

It might seem that 0.8 and 0.9 are very large values that leave no possibility for constraints to be watched. This is definitely not the case. In the original system with threshold 0.8 the median of the percentage of constraints that are watched was 72%, whereas it was 56% in the improved version with threshold 0.9. Since our improvements had a larger impact on counter-based propagation, it is not a surprise that the best improved hybrid version uses counter-based propagation more often than the best original hybrid version.

Finally, we compare in the rightmost plot of Figure 3 the best hybrid methods that use the original and the improved propagation mechanisms, respectively. We can see that, not surprisingly, our improvements on these procedures also result in an improvement on the corresponding hybrid methods.

The last contribution of this paper is to determine whether the improvements we have shown in propagation speed translate into improvements on the overall runtime of RoundingSat. That is, we now run RoundingSat without reading the logs, but rather as a complete PB solver that performs conflict analysis, cleanups, restarts, etc., in an autonomous way, without relying on an external oracle. In Figure 4 we compare the best system using the improved propagation routines, which is the hybrid approach with threshold 0.9 with the best system using the original propagation routines, which is the hybrid approach with threshold 0.8. Results are on all 1600 benchmarks in the category OPT-SMALLINT-LIN with a time limit of 3600 seconds. We believe that the plot is very clear and leaves no doubt that remarkable speedups are obtained thanks to the research presented in this paper.

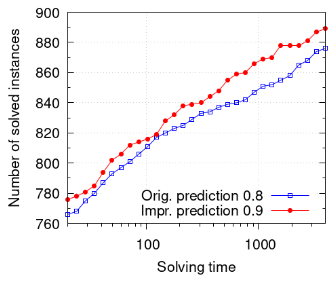


Figure 4 Hybrid method on original and improved propagation. No logs are used.

7 Conclusions and Future Work

We have presented a novel methodology to evaluate propagation procedures and applied it to improve the implementation of these routines inside RoundingSat, currently the fastest CDCL-based PB solver. This has resulted in important speedups in performance.

As future work, we plan to precisely analyze the impact of maintaining (an upper bound on) the maximum coefficient of *undefined* literals in order to have a more precise filter for propagation. Also, computing slacks with respect to the whole assignment, instead of only considering the propagated literals is part of our future work. Additionally, we want to develop hybrid methods that choose between counter or watched-based propagation by dynamically evaluating how active the literals in a constraint are, as opposed to only focusing on their coefficients.

References

1 RoundingSAT web page. `https://gitlab.com/MIAOresearch/software/roundingsat`. Accessed: 2024-03-11.

2 Roberto Javier Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and maxsat. *Ann. Oper. Res.*, 218(1):71–91, 2014.

3 Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 450–457, New York, NY, USA, 2002. ACM. `doi:10.1145/774572.774638`.

4 Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011. `doi:10.1613/jair.3152`.

5 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010. URL: `http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf`, `doi:10.3233/SAT190075`.

6 Armin Biere. Picosat essentials. *JSAT*, 4(2-4):75–97, 2008. URL: `http://jsat.ewi.tudelft.nl/content/volume4/JSAT4_5_Biere.pdf`.

7 Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2010. Technical Report 10/1, August 2010, FMV Reports Series.

8 Armin Biere. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series in Computing*, page 88. EasyChair, 2014. `doi:10.29007/JHD7`.

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

10 Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018. `doi:10.1007/978-3-319-10575-8_10`.

11 Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.

12 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *J. Satisf. Boolean Model. Comput.*, 6(1-3):99–120, 2009. `doi:10.3233/SAT190064`.

13 W. Cook, C. Coullard, and Gy. Turan. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.

14 Jo Devriendt. Watched propagation of 0-1 integer linear constraints. 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 160–176. Springer, 2020. doi:10.1007/978-3-030-58475-7_10.

15 Jo Devriendt, Ambros M. Gleixner, and Jakob Nordström. Learn to relax: Integrating 0-1 integer linear programming with pseudo-boolean conflict-driven search. *Constraints An Int. J.*, 26(1):26–55, 2021. doi:10.1007/S10601-020-09318-X.

16 Jo Devriendt, Stephan Gocht, Emir Demirovic, Jakob Nordström, and Peter J. Stuckey. Cutting to the core of pseudo-boolean optimization: Combining core-guided search with cutting planes reasoning. 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 3750–3758. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16492.

17 Julian Dolby, Mandana Vaziri, and Frank Tip. Finding Bugs Efficiently With a SAT Solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 195–204, 2007. doi:10.1145/1287624.1287653.

18 Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT '03*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

19 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/ijcai.2018/180.

20 Ian P. Gent. Optimal implementation of watched literals and more general techniques. *J. Artif. Intell. Res.*, 48:231–251, 2013. doi:10.1613/JAIR.4016.

21 Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2 & 3):297–308, 1985.

22 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.

23 Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 519–534. Springer, 2010. doi:10.1007/978-3-642-16242-8_37.

24 Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of sat-solvers. In Daniel Le Berre, editor, *POS-10. Pragmatics of SAT, Edinburgh, UK, July 10, 2010*, volume 8 of *EPiC Series in Computing*, pages 28–40. EasyChair, 2010. doi:10.29007/3VWV.

25 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

26 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

27 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi:10.1016/j.artint.2010.10.002.

28 Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in AI and Applications*, pages 695–733. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-695.

29 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A modern pseudo-boolean SAT solver. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 684–685. IEEE Computer Society, 2005. doi:10.1109/DATE.2005.246.

30 Hossein M. Sheini and Karem A. Sakallah. Pueblo: A hybrid pseudo-boolean SAT solver. *JSAT*, 2(1-4):165–189, 2006.

31 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

32 Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005*, pages 139–143, 2005. doi:10.1007/11513988_13.

33 L. Zhang and S. Malik. Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms. In E. Giunchiglia and A. Tacchella, editors, *6th International Conference on Theory and Applications of Satisfiability Testing, SAT '03*, volume 2919 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2004.

eSLIM: Circuit Minimization with SAT Based Local Improvement

Franz-Xaver Reichl
TU Wien, Austria

Friedrich Slivovsky
University of Liverpool, UK

Stefan Szeider
TU Wien, Austria

Abstract

eSLIM is a tool for circuit minimization that utilizes **E**xact **S**ynthesis and the **S**AT-based **l**ocal **i**mprovement **m**ethod (**SLIM**) to locally improve circuits. eSLIM improves upon the earlier prototype CIOPS that uses Quantified Boolean Formulas (QBF) to succinctly encode resynthesis of multi-output subcircuits subject to don't cares. This paper describes two improvements. First, it presents a purely propositional encoding based on a Boolean relation characterizing the input-output behavior of the subcircuit under don't cares. This allows the use of a SAT solver for resynthesis, substantially reducing running times when applied to functions from the IWLS 2023 competition, where eSLIM placed second. Second, it proposes circuit partitioning techniques in which don't cares for a subcircuit are captured only with respect to an enclosing window, rather than the entire circuit. Circuit partitioning trades completeness for efficiency, and successfully enables the application of exact synthesis to some of the largest circuits in the EPFL suite, leading to improvements over the current best implementation for several instances.

2012 ACM Subject Classification Hardware → Circuit optimization; Theory of computation → Automated reasoning

Keywords and phrases QBF, Exact Synthesis, Circuit Minimization, SLIM

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.23

Supplementary Material *Software*: `https://github.com/fxreichl/eSLIM` [29]
archived at `swh:1:dir:9d2862aedfe458cfb06d4d43da6c0f46ed578c91`

Funding This work was supported by the Vienna Science and Technology Fund (WWTF) under grant 10.47379/ICT19060, and the Austrian Science Fund (FWF) under grants 10.55776/W1255 and 10.55776/P36420.

Acknowledgements The authors thank Alan Mishchenko for suggesting to represent local synthesis tasks with Boolean relations.

1 Introduction

Modern integrated circuits have grown increasingly large and complex, making their design and optimization a significant challenge. Automation has become indispensable for the process of circuit design, including logic optimization and logic synthesis, which collectively lead to substantial reductions in the number of gates and circuit depth [6, 10].

Applying exact methods for computing provably minimum size circuits is computationally intractable. Recently, it was shown that for a (multi-output) Boolean function given as a truth table, the task of finding a minimum size circuit consisting of *and*, *or* and *not* gates is NP-complete [17]. This is reflected by the observation that in practice, we can generally not compute minimum size circuits with many more than 10 fanin-2 gates [9, 21]. To deal with larger circuits, one can first partition them into smaller subcircuits, and then minimize these using exact techniques [39].

In practice, this approach is typically restricted to single-output subcircuits [25, 31]. Although efficient, this does not fully exploit the implementation flexibility of multi-output subcircuits. Recently, we proposed a high-effort method for resynthesizing multi-output subcircuits based on Quantified Boolean Formulas (QBFs) [30]. Our prototype CIOPS has shown success in minimizing circuits from the IWLS'22 competition and the EPFL combinational benchmark suite. Specifically due to increasing prices of silicon wafers in recent years [15] such high-effort methods are gaining importance.

In this paper, we describe two improvements for this approach, which we implemented in the new tool eSLIM. First, we present a workflow purely based on SAT instead of QBF. The new SAT workflow relies on computing an input-output relation that yields permissible output assignments for each input assignment. These relations allow representing *don't cares* – input/output combinations of the subcircuit where the outputs can be modified without altering the function computed by the encompassing circuit. In contrast, the QBF encoding captures don't cares implicitly. Our rationale is that the ability to use a SAT solver instead of a QBF solver will more than make up for this extra step when the relation is reasonably small. Second, we incorporate windowing to handle very large circuits. For such circuits, the QBFs encoding the existence of replacements for subcircuits become too hard for QBF solvers, and computing the input-output relation needed for the new SAT encoding takes too much time. To address this, we adapt a strategy from prior work on computing don't cares of single-output subcircuits [25]. Instead of ensuring that a resynthesized subcircuit preserves the Boolean function computed by the full circuit, we only require that it preserves the function computed by a "window" containing the subcircuit to be replaced. In theory, this means don't cares of a subcircuit are no longer fully captured. In practice, with windows containing hundreds of gates, we expect that generally don't cares within the window still allow substantial improvements.

We performed an experimental evaluation of these improvements. The SAT-based workflow proved to be substantially faster for small circuits, showing a significant performance increase for instances from the IWLS'23 competition. For circuits from the EPFL combinational benchmark suite, the SAT and the QBF-based approach showed comparable performance. Using windowing, we were able to scale both approaches to the largest circuits in this set, which had previously been unmanageable.

1.1 Related Work

Methods that fully capture the properties of Boolean functions implemented by circuits (rather than considering them as polynomials, for instance) are deemed the most effective in logic synthesis [39]. However, these methods are also the most computationally expensive, and their application to large circuits is limited to resynthesizing small subcircuits. SAT-based exact synthesis [16, 21] and SAT-based resubstitution [26, 32], which aim to represent the function implemented by a specific gate as a function of a few existing gates in the circuit, are examples of this approach. Many of these methods are implemented in the industrial-strength tool ABC [7].

A SAT-based method for capturing don't cares close to ours has been previously considered for single-output subcircuits, including the use of windowing to improve scalability [25]. Boolean relations have been proposed as a means of representing don't cares of multi-output subcircuits [23]. The corresponding optimization workflow relies on a simulation-based under-approximation of don't cares in combination with a divide-and-conquer algorithm for resynthesis, whereas we use a SAT solver for both don't care computation and resynthesis.

Exact resynthesis of subcircuits has also been explored for finding optimal circuits for symmetric functions in circuit complexity research, but without incorporating don't cares [22]. This is an instance of the *SAT-based Local Improvement Method (SLIM)*, a general optimization framework that has been applied to various AI problems [11, 24, 27, 34, 35, 36].

In the realm of logic synthesis, QBFs have been employed for bi-decomposition [8], reversible quantum circuit synthesis [41], and lookup table (LUT) synthesis [12, 13, 14]. The latter two problems impose more constraints than the setting considered in this work, as they maintain a fixed circuit topology. In contrast, our synthesis tasks also involve determining a suitable topology.

2 Preliminaries

A *Boolean circuit* is a *directed acyclic graph*. We denote the set of source nodes of a circuit $\mathcal{C}$ as the set of *primary inputs* $\text{in}(\mathcal{C})$ and the non-source nodes as *gates*. The set of *primary outputs* $\text{out}(\mathcal{C})$ is a subset of the set of nodes in $\mathcal{C}$. If there is a directed edge from node n to node m then n is an *input* of m. Each gate corresponds to a Boolean function on its inputs. This means that each Boolean circuit naturally induces a Boolean function. Let x be a node in $\mathcal{C}$. Then the *transitive fanin cone* of x, $\mathit{TFI}(x)$ is the set of all nodes in $\mathcal{C}$ from which x is reachable. Similarly, the *transitive fanout cone* of x $\mathit{TFO}(x)$ is the set of all nodes in $\mathcal{C}$ which can be reached from x. A node x *depends* on a node y if $y \in \mathit{TFI}(x)$. Two circuits $\mathcal{C}$ and $\mathcal{D}$ are logically equivalent ($\mathcal{C} \equiv \mathcal{D}$) if they compute the same Boolean function. *Don't cares* of a subcircuit $\mathcal{S}$ of a circuit $\mathcal{C}$ are certain patterns of its inputs or outputs that do not have an effect on the function computed by the encompassing circuit $\mathcal{C}$. We distinguish between two types of don't cares. *Controllability don't cares* are patterns of the inputs of $\mathcal{S}$ which cannot be attained within $\mathcal{C}$, and *observability don't cares* are patterns of the inputs of $\mathcal{S}$ for which its outputs do not have an effect on $\mathcal{C}$ [10].

A *Quantified Boolean formula (QBF)* is of the form $\forall X_1 \exists X_2 \ldots \forall X_{k-1} \exists X_k . \varphi$, where the X_i are pairwise disjoint sets of variables, and φ is a propositional formula called the *matrix*. The quantifiers range over the Boolean domain $\mathcal{B} = \{0, 1\}$, so that existential ($\exists$) quantifiers can be understood as abbreviating a disjunction ($\exists x.\varphi \equiv \varphi[x \leftarrow 0] \vee \varphi[x \leftarrow 1]$), and universal ($\forall$) quantifiers as encoding a conjunction ($\forall x.\varphi \equiv \varphi[x \leftarrow 0] \wedge \varphi[x \leftarrow 1]$). Evaluating QBFs is a PSPACE-complete task, and QBFs can succinctly encode problems arising in many areas [37]. For an overview of QBF, including solving techniques and proof complexity, see [2].

3 Exact Synthesis of Subcircuits

Our approach involves replacing a subcircuit with a smaller one, ensuring that the function computed by the encompassing circuit remains unchanged. For a fixed value ℓ, we use a SAT or QBF solver to determine whether there is a replacement circuit of size ℓ. To find a smallest possible circuit, the value ℓ is decremented until the encoding becomes unsatisfiable.

Throughout this section, let $\mathcal{C}$ denote the encompassing circuit, $\mathcal{S}$ one of its subcircuits, and $\mathcal{T}$ the replacement circuit. Further, n is the number of inputs of the subcircuit $\mathcal{S}$ and m the number of outputs. Moreover, we assume that $\mathcal{C}$ is *k-regular*, i.e., each gate in $\mathcal{C}$ has k inputs. Let $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ denote the result of substituting the circuit $\mathcal{T}$ for the subcircuit $\mathcal{S}$ in $\mathcal{C}$. Our goal is to find a circuit $\mathcal{T}$ of size ℓ with n inputs and m outputs such that $\mathcal{C}$ and $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$ are logically equivalent. This means that $\mathcal{T}$ and $\mathcal{S}$ do not necessarily need to be equivalent. In particular this also means that the solver can assign an arbitrary behavior to $\mathcal{T}$ for input/output combinations of the subcircuit that do not have an influence on the entire circuits – i.e., we can make use of don't cares implicitly.

We initially set $\ell = |\mathcal{S}|$ and then decrement ℓ until the solver determines that no such circuit $\mathcal{T}$ exists, at which point we can conclude that the circuit $\mathcal{T}$ must have at least $\ell + 1$ gates. A circuit $\mathcal{T}$ of this size can be constructed from a model of the last satisfiable encoding, and used to replace $\mathcal{S}$. It may seem unnecessary to initially ask the solver to come up with a circuit of size $\ell = |\mathcal{S}|$, since we already know such a circuit exists. However, the new circuit is typically not equivalent to $\mathcal{S}$, and replacing $\mathcal{S}$ by $\mathcal{T}$ is often beneficial in the overall minimization process even though it does not immediately decrease size [30].

Subsequently, we will describe both the QBF and the SAT encoding. Both are closely related to and inspired by the *multi selection variable* SAT encoding for exact synthesis [16].

3.1 The QBF Encoding

The main advantage of a QBF encoding is that one can universally quantify over assignments of primary inputs and encode the value computed by each gate using a single variable, rather than introducing a variable for each line in the truth table representation of $\mathcal{C}$. In addition to these **gate variables** and primary inputs, the encoding contains the following groups of existentially quantified variables that determine the structure of the circuit:

Selection variables $S_i = \{s_{it} \mid 1 \le t < i + n\}$. This set of variables determines the inputs of the i^{th} gate. If s_{it} is true then the t^{th} node is an input of the i^{th} gate[1].

Function variables $F_i = \{f^i_{a_1 \ldots a_k} \mid (a_1, \ldots, b_k) \in \mathcal{B}^k\}$. This set of variables describes the Boolean function f at the i^{th} gate, i.e., the assignment of $f^i_{a_1 \ldots a_k}$ determines the value of $f(a_1, \ldots, a_k)$.[2]

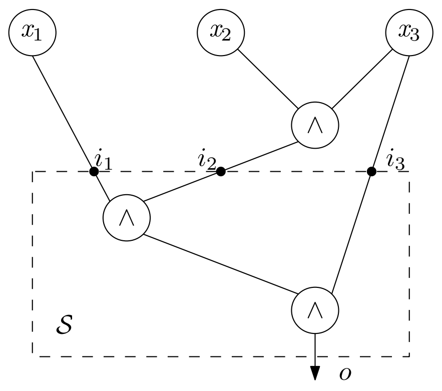
Output variables $O_j = \{o_{tj} \mid 0 \le t \le n + \ell\}$. This set of variables fixes the output gates of the circuit. If o_{tj} is true then the j^{th} output is given by the constant value false if $t = 0$ and otherwise by the t^{th} node.[2]

The matrix of the QBF encodes the following constraints:

- Each gate must have exactly k inputs, i.e., at each gate i exactly k selection variables must be true. This can be enforced by using a sequential counter [38]. We denote this constraint by $\mathit{Count}(S_i, k)$.
- Each output must correspond to a single gate, i.e., for each output j exactly one output variable is true. We denote this constraint by $\mathit{Count}(O_j, 1)$.
- For each gate i the assignment of the gate variable must be compatible with the function determined by the function variables. Assume that the assignment for the function variables describes the function F and that the values of the inputs of i are given by $i_1, \ldots, i_k$. This constraint ensures that the gate variable for i is assigned to $F(i_1, \ldots, i_k)$. We denote this constraint by Comp_i.
- Replacing $\mathcal{S}$ by $\mathcal{T}$ must preserve the function computed by $\mathcal{C}$. To express this, one can use Tseitin transformation and two sets of gate variables to encode both the specification $\mathcal{C}$ and the circuit $\mathcal{C}' = \mathcal{C}[\mathcal{S} \leftarrow \mathcal{T}]$. For an output gate o, let v_o and v'_o denote the gate variable in the encoding of $\mathcal{C}$ and $\mathcal{C}'$, respectively. We add a constraint $v_o \Leftrightarrow v'_o$ and denote their conjunction over all outputs by Corr.

[1] We index gates from 1 to ℓ, and we index nodes from 1 to $n + \ell$. The nodes with indices from 1 to n correspond to the primary inputs. For $n < i \le n + \ell$ the i^{th} node corresponds to the $(i - n)^{\text{th}}$ gate.

[2] A Boolean function is *normal* if $f(0, \ldots, 0) = 0$, and a circuit is *normal* if each of its gates corresponds to a normal function. In a normal circuit, we can find minimum size normal replacement circuits [20]. We can always normalize a given circuit, reduce the size of the normalized circuit, and finally we can reconstruct the original circuit by flipping output gates if necessary. Thus, $f^i_{0 \ldots 0}$ can be set to false, and outputs yielding the constant value true don't need to be considered.



i_1	i_2	i_3	o
0	0	0	$\{0\}$
0	0	1	$\{0\}$
0	1	0	$\{0,1\}$
0	1	1	$\{0\}$
1	0	0	$\{0\}$
1	0	1	$\{0\}$
1	1	0	$\{0,1\}$
1	1	1	$\{1\}$

Figure 1 The table on the right gives a Boolean relation for the subcircuit $\mathcal{S}$. One can see that the input pattern $i_2 = 1, i_3 = 0$ is not possible within the entire circuit. The relation given in the table makes use of this controllability don't care by assigning $\{0,1\}$ to the corresponding assignments of the inputs.

Let $S = \bigcup_{1 \leq i \leq k} S_i$ denote the set of selection variables, F the set of gate definition variables, O the set of output variables, I the set of input variables, and G, G' the two sets of gate variables. The complete QBF encoding has the following form:

$$\exists S, F, O\, \forall I\, \exists G, G'.\ \mathit{Corr} \wedge \bigwedge_{1 \leq j \leq m} \mathit{Count}(O_j, 1) \wedge \bigwedge_{1 \leq i \leq \ell} (\mathit{Count}(S_i, k) \wedge \mathit{Comp}_i)$$

In practice, we use the circuit-based QCIR format for the encoding. Thus, we can directly define auxiliary variables by QCIR-gates and so we do not need to add these variables to the prefix.

3.2 The SAT Encoding

While the QBF encoding handles don't cares implicitly, the SAT-based approach outlined in this subsection separates the tasks of computing don't cares and synthesizing a subcircuit. More specifically, it first computes a *Boolean relation* [6, 33] representing the input-output behavior of the subcircuit $\mathcal{S}$ on the *care set* (the complement of don't cares), and then uses the *multi selection variable* SAT-encoding [16] to obtain a circuit.

A Boolean relation R for $\mathcal{S}$ maps each input assignment to a set of permissible output assignments, formally $R : \mathcal{B}^{\mathrm{in}(\mathcal{S})} \to (\mathcal{P}(\mathcal{B}^{\mathrm{out}(\mathcal{S})}) \setminus \emptyset)$. A circuit C *implements* the relation R if $C(\sigma) \in R(\sigma)$ for each $\sigma \in \mathcal{B}^{\mathrm{in}(C)}$. In Figure 1 we illustrate a Boolean relation representing a subcircuit $\mathcal{S}$ of a circuit $\mathcal{C}$ that takes the don't cares of $\mathcal{S}$ into account.

The core idea for computing the relation is to determine the set $\Phi \subset \mathcal{B}^{\mathrm{in}(\mathcal{S})} \times \mathcal{B}^{\mathrm{out}(\mathcal{S})}$ of conflicting input/output behaviors. A conflicting input/output behavior is a pair of assignments (σ, ρ), where σ is an assignment of $\mathcal{B}^{\mathrm{in}(\mathcal{S})}$ and ρ is an assignment of $\mathcal{B}^{\mathrm{out}(\mathcal{S})}$. For each such pair we require that for any circuit $\mathcal{D}$ with inputs $\mathrm{in}(\mathcal{S})$, outputs $\mathrm{out}(\mathcal{S})$ and $\mathcal{D}(\sigma) = \rho$, we have $\mathcal{C}[\mathcal{S} \leftarrow \mathcal{D}] \not\equiv \mathcal{C}$. This means Φ describes all modifications of $\mathcal{S}$ that would change the function computed by $\mathcal{C}$. The relation R is then given by $R(\sigma) = \{\rho \in \mathcal{B}^{\mathrm{out}(\mathcal{S})} \mid (\sigma, \rho) \notin \Phi\}$. One can easily verify that replacing $\mathcal{S}$ by any circuit implementing this relation preserves the function computed by the entire circuit.

In order to obtain Φ, we first compute a circuit $\mathcal{C}'$ by removing $\mathcal{S}$ from $\mathcal{C}$.[3] Due to the removal of gates, $\mathcal{C}'$ contains an additional primary input for each output of $\mathcal{S}$. We denote the set of these new inputs by $\mathcal{I}$. Next, we compute every assignment σ for $\mathrm{in}(\mathcal{C})$ and ρ for $\mathcal{I}$ such that $\mathcal{C}(\sigma) \neq \mathcal{C}'(\sigma \cup \rho)$. Now let σ' be the assignment of the inputs of $\mathcal{S}$ ($\mathrm{in}(\mathcal{S})$), which is attained by $\mathcal{C}$ under σ. Then Φ is the set of all pairs (σ', ρ).

To realize this idea we first compute clausal encodings φ_1 for $\mathcal{C}$ and φ_2 for $\mathcal{C}'$ by Tseitin transformation, introducing a propositional variable for each node. For each node x in $\mathcal{C}$ we denote the corresponding variable by $v(x)$ and for each node x in $\mathcal{C}'$ by $v'(x)$. Similarly, we define v / v' for sets of nodes. Next we introduce for each common primary output o the constraint $v(o) \Leftrightarrow v'(o)$, denoting the set consisting of all these clauses by *equiv*.

This encoding can now be used to compute the relation with incremental SAT solving. The algorithm maintains a set B of blocking clauses, which is empty initially. We now ask the SAT solver for an assignment σ that satisfies $\varphi_1 \wedge \varphi_2 \wedge B$ and falsifies *equiv*. Instead of directly adding $\neg\sigma|_{v(\mathrm{in}(\mathcal{S}))} \vee \neg\sigma|_{v'(\mathcal{I})}$ to the clausal representation of the relation, we first try to reduce this clause. For this purpose, we apply a similar approach as it was used by Ravi and Somenzi [28] for reducing the size of assignments. The formula $\varphi_1 \wedge \varphi_2 \wedge \mathit{equiv}$ is unsatisfiable under the assumption $\sigma|_{v(\mathrm{in}(\mathcal{C}))} \wedge \sigma|_{v'(\mathcal{I})}$. Using the SAT solver, we can compute a subset $\hat{\sigma}$ of failed assumptions. As $\hat{\sigma}$ suffices to make $\varphi_1 \wedge \varphi_2 \wedge \mathit{equiv}$ unsatisfiable it is rather easy to conclude that for each assignment μ with $\hat{\sigma}|_{v'(\mathcal{I})} \subseteq \mu$, the pair $(\sigma|_{v(\mathrm{in}(\mathcal{S}))}, \mu)$ must not be contained in the relation. Thus, we can add $\neg\sigma|_{v(\mathrm{in}(\mathcal{S}))} \vee \neg\hat{\sigma}|_{v'(\mathcal{I})}$ to the clausal representation of the relation. Moreover, to avoid the same inconsistency in subsequent iterations, we add the blocking clause $\neg\sigma|_{v(\mathrm{in}(\mathcal{S}))} \vee \neg\hat{\sigma}|_{v'(\mathcal{I})}$ to B.

As mentioned above, a circuit implementing the relation R can be synthesized using a slight adaptation of the SAT encoding for exact synthesis by Haaswijk et al. [16].

4 Minimization by Subcircuit Resynthesis

We use exact synthesis of subcircuits as a subroutine in a circuit minimization algorithm that repeatedly selects subcircuits for resynthesis. To obtain a subcircuit for resynthesis, we start from a *root gate*, and then expand by incorporating successors of previously chosen gates until reaching a predetermined size. Root gates are chosen randomly from the circuit. To expand the root gate, we visit gates that use previously selected gates as inputs in a breadth-first-search manner. We then randomly decide whether to include this gate in the subcircuit. Unlike previous work [30], we use a fixed bound for the size of subcircuits. This bound is decreased in case individual checks timeout.

5 Window Selection

Both the SAT and the QBF-based rewriting approach do not only depend on the selected subcircuits but also on the entire circuit. This is necessary to make use of don't cares. Thus, in general, rewriting subcircuits gets harder for larger circuits. To overcome this issue, we only consider don't cares with respect to a *window* (a subcircuit) [25]. The window is chosen such that the size is still manageable for our QBF/SAT-based approach. In this manner, we can minimize windows, and since the functions computed by windows are preserved, the optimized implementation can be used to replace the original window.

[3] Actually, it suffices to consider $\mathit{TFO}(\mathcal{S})$ instead of a copy of the entire circuit

Additionally, using windows allows us to rewrite subcircuits simultaneously. For this purpose, we compute pairwise disjoint windows and rewrite them separately. In the end, we then combine the optimized windows to obtain a new implementation for the original circuit.

6 Implementation

We implemented eSLIM mainly in Python and some parts in C++. For using C++ subroutines in Python, we used PYBIND11[4]. As backend solvers, we used QFUN [18] for QBF and CaDiCaL [4] for SAT. To read and write files in the *AIGER* format, we used the AIGER library [5]. To represent arrays of Boolean values, we used the python library BITARRAY[5]. eSLIM is freely available[6] under an MIT license.

Input/Output Formats

eSLIM can process circuits given in the Berkeley Logic Interchange Format (BLIF) [40] and in the AIGER format [3]. The given circuits have to be purely combinational and thus must not contain any sequential components. Circuits in the BLIF format are required to be sorted topologically, i.e., fanins of a gate must be specified before the gate itself. For the AIGER format, both the ASCII and the binary AIGER format are supported. Similar as for circuits given in the BLIF format, also ASCII AIGER circuits are expected to be sorted topologically. The main output format of eSLIM is BLIF. In case an AIG shall be reduced, also the binary and ASCII AIGER format are supported.

Parameters of eSLIM

Our tool can be applied as follows.

```
eSLIM <circuit> <result> <budget> [options]
```

We first describe the mandatory arguments:
circuit The circuit to be processed in one of the supported formats.
result The destination for the minimized circuit (given in the BLIF format).
budget The available time in seconds.
Additionally, eSLIM accepts several optional arguments. We only list the most important ones, and we refer to the help information provided by eSLIM for the remaining options.
--**gs** The number of fanins of the synthesized gates.
--**size** The upper bound for considered subcircuit sizes.
--**aig** Synthesize an AIG. This option requires that the fanin size is set to two.
--**aig-out** Write the minimized circuit to a binary or ASCII AIGER file.
--**restarts** Specify the number of applications of our method. Each application can use `budget` seconds.
--**abc** Use ABC for inprocessing after each application of our method.
--**syn-mode** Specify whether the QBF- or the SAT-based approach shall be used.
--**windows** Specify the number of windows that shall be processed concurrently and the reference size for each considered windows.

[4] `https://github.com/pybind/pybind11`
[5] `https://pypi.org/project/bitarray/`
[6] `https://github.com/fxreichl/eSLIM`

Types of Logic Gates

In its default configuration, eSLIM computes replacement circuits composed of arbitrary k-fanin gates – where k is specified by `--gs`. The set of available functions for each gate can be constrained by restricting the assignments for the function variables. This, for example, means that if the option `--aig` is used, then XOR gates are ruled out by requiring $\neg f^i_{01} \vee \neg f^i_{10} \vee f^i_{11}$ for each $1 \leq i \leq \ell$, where ℓ denotes the considered circuit size. In addition to computing AIGs, we also used eSLIM to compute XAIGs – AIGs that may contain XOR gates. By adding appropriate constraints for the function variables, additional restrictions can be placed on the set of available functions.

7 Experiments

We evaluated our tool on the instances from the IWLS'23 programming contest[7] and the instances from the EPFL benchmark suite [1]. All experiments were conducted on a cluster with AMD EPYC 7402 processors at 2.8 GHz running 64-bit Linux. We used a memory limit of 4 GB. For the parallelized minimization we used a memory limit of 4 GB per thread.

7.1 IWLS Instances

The IWLS'23 instances consist of 100 instances, given as truth tables. The goal is to compute an *And-Inverter Graph* (AIG) with as few gates as possible. A preliminary version of eSLIM participated in the competition and took the second place.

Since the instances are given as truth tables, and our tool requires that specifications are given as circuits, we had to preprocess the instances using ABC [7]. As a naive transformation of truth tables to circuits by using ABC results in relatively large circuits, we used ABC to reduce the size of the initial circuit. For this purpose, we used the ABC command DEEPSYN with a timeout of one hour[8]. We chose DEEPSYN as it is one of the most effective optimization strategies for computing compact AIGs [9]. Additionally, unlike many other optimization strategies in ABC, DEEPSYN allows any-time optimization.

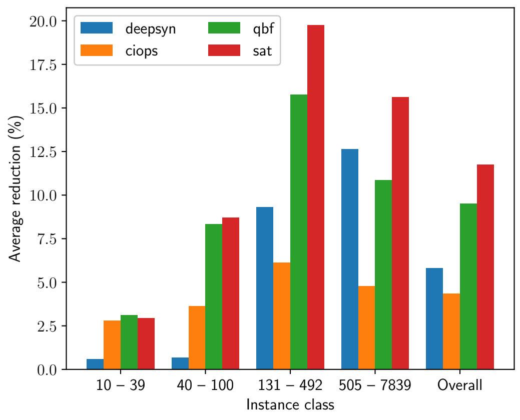
In our evaluation setup, we considered our tool both in the QBF-based and the SAT-based configuration. As only very few of the IWLS instances are sufficiently large for a reasonable application of windowing, we did not evaluate it here. We compared eSLIM with the prototype of the purely QBF-based approach CIOPS [30] and the DEEPSYN procedure[9].

In our experiments, we alternated between 27-minute runs of eSLIM and 3-minute runs of DEEPSYN for inprocessing. Similarly, we alternated between 30-minute runs of CIOPS and exhaustive heuristic minimization with ABC. Here *exhaustive minimization* means that the application of ABC is repeated until no further reductions can be found. We wanted to compare eSLIM with the original version of CIOPS, thus we used different setups for inprocessing in eSLIM and CIOPS. These combinations were applied eight times. We used 30-minute runs of CIOPS as the time needed for inprocessing cannot be controlled. As inprocessing usually only takes a few seconds, still in total roughly 4 hours were available per instance. Finally, DEEPSYN was run for four hours. In general, all tools benefited from

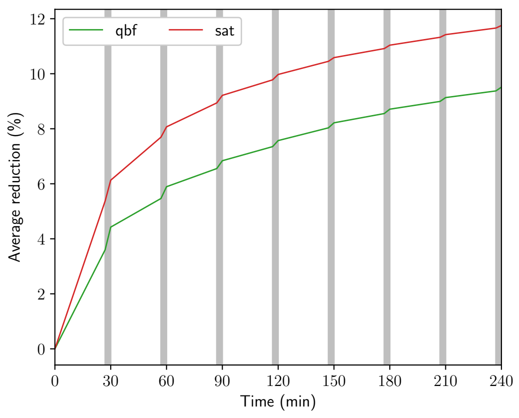
[7] `https://github.com/alanminko/iwls2023-ls-contest`
[8] To run DEEPSYN we used `&deepsyn -T 3600`. By using `-T 3600` we enforced the timeout.
[9] We could not compare our tool against the first and third ranked entry to the IWLS 2023 competition – our tool ranked second – as they were not available to us.

Table 1 Average reduction (%) of gates compared to the preprocessed IWLS'23 instances, by configuration and initial size and standard deviations of the average reductions per configuration. The best results are marked in boldface.

	CIOPS		DEEPSYN		QBF		SAT	
#Gates	mean	stdev	mean	stdev	mean	stdev	mean	stdev
10-39	2.8	0.28	0.59	0.4	**3.11**	0.19	2.93	0.28
40-100	3.64	0.19	0.68	0.18	8.34	0.43	**8.7**	0.26
131-492	6.12	0.22	9.32	0.44	15.75	0.57	**19.76**	0.59
505-7839	4.79	0.13	12.64	0.47	10.85	0.39	**15.61**	0.52
Overall	4.34	0.11	5.82	0.22	9.51	0.25	**11.75**	0.17



(a) Average reductions.



(b) Reductions over time.

Figure 2 The left figure (a) visualizes the average reductions of gates per instance class and configuration. The right figure (b) visualizes the average reductions among all instances for the SAT and the QBF configuration over time. The areas marked in gray correspond to the application of ABC.

longer runtimes. Nevertheless, we limited the runs to roughly four hours due to constraints on the available computational infrastructure. Additionally, we set the initial bound for the subcircuit size to 6 both for our tool and CIOPS (cf. Section 4).

Instances were grouped into four subsets of 25 based on the initial number of gates. For each configuration and instance group, we determined the average size reduction (in %) for circuits in that group. We performed 5 independent runs for each configuration. First, we calculated the average reduction per run for each subset and for the entire set of instances. Based on these values, we computed average reductions and standard deviations among the individual runs of each configuration for each class of instances. Results are given in Table 1. A visualization of the results is given in Figure 2a.

The experiments show that eSLIM with the QBF configuration clearly outperformed CIOPS. This was mainly due to two simple but apparently effective changes. First, eSLIM uses fixed bounds for the sizes of selected subcircuits, while CIOPS tries to increase the initially given bounds as far as possible. Since larger circuits are usually harder to analyze, this indicates that, in general, it is advantageous to consider more but simpler (smaller) circuits. Second, eSLIM computes subcircuits by expanding root gates in a randomized breadth-first-search manner, while CIOPS applies an expansion strategy that aims at keeping the number of outputs low.

Moreover, the SAT-based strategy outperformed the QBF-based strategy. This was possible as in general the SAT-based approach allowed a faster analysis of subcircuits. Thus, more subcircuits could be analyzed in total. Furthermore, eSLIM could outperform DEEPSYN. This indicates that the combination of DEEPSYN for preprocessing/inprocessing with eSLIM is viable alternative to just applying DEEPSYN alone.

Furthermore, to illustrate the achieved reductions over time we give Figure 2b. The figure shows the average reduction among all instances for the QBF- and the SAT-based configuration over time. We can see that the achieved improvements diminish the longer the tool ran, still on average both approaches could find new improvements until the end. For the QBF- and the SAT-based configuration the DEEPSYN inprocessing steps were responsible for 26%, respectively 18% of the total reductions.

Additionally, if we consider the best implementations for each instance among the 5 SAT runs, we can observe an average reduction of about 14%. This indicates that it may be advantageous to consider multiple runs of eSLIM. As our tool makes use of a randomized subcircuit selection, different runs result in different sequences of replaced subcircuits. Thus, a run might get stuck in a local minimum, which is difficult to escape.

7.2 EPFL Instances

To evaluate our tool for circuits with non-binary gates, we considered the *EPFL Combinational Benchmark Suite* [1]. This benchmark set consists of twenty circuits.[10] The goal is to find 6-input lookup table (LUT-6) implementations of the specifications with small size. In addition to a specification given as a circuit with binary gates, the benchmark suite also provides the best known LUT-6 realizations so far. We used the best known realizations as of 2022 (commit *42c1f31*) as initial specifications for our tool.[11]

We ran our reduction tool for 12 hours both with the SAT and the QBF configuration. After each hour we applied the ABC command &MFS as an inprocessing step – we used &MFS as it allows us to directly optimize an LUT-6 circuit. Additionally, we also applied our tool with windowing enabled. Here, we recombined the windows for the inprocessing step and computed new windows afterwards. We compared our tool with CIOPS. We want to point out that the initial realizations have already been highly optimized by different methods, so any improvement can be considered a success.

In addition to a bound on the size of subcircuits, we also used a limit of 10 on the number of inputs of the subcircuits considered for resynthesis. Preliminary tests showed that such a limit is required to reliably generate the Boolean relation for the SAT-based approach within time and memory limits. Additionally, we always set the initial bound for the subcircuit size to 4. In the experiments with windowing, we used two different window sizes, 500 and 1000. First we minimized single windows and second up to 8 windows concurrently. We only applied windowing for instances with at least 1000 gates.

Results for instances with less than 1000 gates are given in Table 2a and results for instances with at least 1000 gates are given in Table 2b. For the results with windowing we selected the best results among the two different window sizes.

Since the initial circuits are already highly optimized by state-of-the-art methods, the relative improvements for the EPFL instances were small compared to the IWLS instances, and it is difficult to draw any definitive conclusion about the superiority of any configuration

[10] We did not consider the MtM instances as the EPFL repository does not contain the best implementations for these circuits.

[11] We did not consider the best results of 2023 as half of them were provided by us. As it is difficult for our tool to further reduce these circuits, we think the circuits from 2022 are better suited for the evaluation.

Table 2 The table gives for each instance the number of LUT-6 gates of the initial circuit and of the improved circuits per configuration. The best results are marked in boldface.

(a) Results for EPFL instances with less than 1000 gates.

Instance	Initial	CIOPS	QBF	SAT
Lookahead XY router	**19**	19	19	19
int to float converter	20	20	**18**	19
Alu control unit	**25**	25	25	25
Coding-cavlc	54	52	**49**	53
Priority encoder	94	94	93	**92**
Adder	**129**	129	129	129
I2c controller	182	178	179	**177**
Decoder	**264**	264	264	264
Round-robin arbiter	273	273	272	**267**
Max	**511**	511	511	511
Barrel shifter	**512**	512	512	512

(b) Results for EPFL instances with fewer than 1000 gates.

			No Windowing		Single Window		Up to 8 Windows	
Instance	Initial	CIOPS	QBF	SAT	QBF	SAT	QBF	SAT
Sine	1114	1111	1095	1085	1076	1069	1057	**1036**
Voter	1217	1217	1217	1179	1184	**1166**	1177	1172
Memory controller	1735	1731	**1722**	1727	1731	1730	1724	1731
Square-root	2994	2994	2994	2994	2991	2985	2992	**2980**
Square	3018	3018	3014	2997	2992	2994	**2942**	2943
Divisor	3096	3096	3096	3096	3096	**3095**	3096	**3095**
Multiplier	4360	4360	4358	4346	4346	4346	4326	**4317**
Log2	6133	6133	6132	6109	6127	6129	6078	**6063**
Hypotenuse	39452	39452	39452	39452	39230	39251	**38459**	38781

from these results. Nevertheless, the results suggest that parallel optimization was able to beat single-threaded optimization. Similarly, the results indicate that both configurations of our tool outperformed CIOPS. Moreover, our tool could improve on the best implementation for the majority of instances.

8 Conclusion

The experimental analysis shows that eSLIM significantly improves upon CIOPS and that eSLIM is a viable alternative to DEEPSYN. Additionally, the experiments show that the SAT-based approach outperforms the QBF-based version for instances from the IWLS'23 programming contest. For circuits from the EPFL suite, the two variants perform very similarly, with either approach having a slight edge in some instances. Moreover, the experiments show that using windowing allows us to further reduce the largest circuits from the EPFL suite.

The respectable performance of the QBF-based approach on larger circuits hints at the potential of adopting techniques from QBF solving, such as counterexample-guided expansion [19]. Computing the entire Boolean relation upfront can be prohibitive, and generating constraints during the substitution process could be more efficient. Specifically, constraints could be added on-the-fly when the substitution of a synthesized circuit alters the function.

References

1 Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *International Workshop on Logic & Synthesis (IWLS)*, 2015. URL: `https://github.com/lsils/benchmarks`.

2 Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified Boolean formulas. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1177–1221. IOS Press, 2021.

3 Armin Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

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

5 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

6 Robert K. Brayton, Gary D. Hachtel, and Alberto L. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proc. IEEE*, 78(2):264–300, 1990.

7 Robert K. Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, volume 6174 of *LNCS*, pages 24–40. Springer, 2010. `doi:10.1007/978-3-642-14295-6_5`.

8 Huan Chen, Mikolás Janota, and João Marques-Silva. QBF-based Boolean function bi-decomposition. In *DATE*, pages 816–819. IEEE, 2012. `doi:10.1109/DATE.2012.6176606`.

9 Andrea Costamagna, Alan Mishchenko, and Giovanni De Micheli. The combinational-complexity game for symmetric functions. In *IWLS*, 2023. URL: `https://si2.epfl.ch/~demichel/publications/archive/2023/AC.pdf`.

10 Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw Hill, 1994.

11 Johannes K. Fichte, Neha Lodha, and Stefan Szeider. SAT-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *LNCS*, pages 401–411. Springer, 2017. `doi:10.1007/978-3-319-66263-3_25`.

12 Masahiro Fujita. Toward unification of synthesis and verification in topologically constrained logic design. *Proc. IEEE*, 103(11):2052–2060, 2015. `doi:10.1109/JPROC.2015.2476472`.

13 Masahiro Fujita, Satoshi Jo, Shohei Ono, and Takeshi Matsumoto. Partial synthesis through sampling with and without specification. In *ICCAD*, pages 787–794. IEEE, 2013. `doi:10.1109/ICCAD.2013.6691203`.

14 Masahiro Fujita, Yusuke Kimura, Xingming Le, Yukio Miyasaka, and Amir Masoud Gharehbaghi. Synthesis and optimization of multiple portions of circuits for ECO based on set-covering and QBF formulations. In *DATE*, pages 744–749. IEEE, 2020. `doi:10.23919/DATE48585.2020.9116459`.

15 Dale Gai. TSMC price hike indicates capacity tightness to persist in 2022, may hit smartphone shipments. online, 2021. URL: `https://www.counterpointresearch.com/insights/tsmc-price-hike/`.

16 Winston Haaswijk, Mathias Soeken, Alan Mishchenko, and Giovanni De Micheli. SAT-based exact synthesis: Encodings, topology families, and parallelism. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(4):871–884, 2020. `doi:10.1109/TCAD.2019.2897703`.

17 Rahul Ilango, Bruno Loff, and Igor C. Oliveira. NP-hardness of circuit minimization for multi-output functions. In Shubhangi Saraf, editor, *CCC*, volume 169 of *LIPIcs*, pages 22:1–22:36. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPICS.CCC.2020.22`.

18 Mikolás Janota. Towards generalization in QBF solving via machine learning. In *AAAI*, pages 6607–6614. AAAI Press, 2018. doi:10.1609/AAAI.V32I1.12208.

19 Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016. doi:10.1016/j.artint.2016.01.004.

20 Donald Ervin Knuth. *The Art of Computer Programming. Volume 4A, Combinatorial Algorithms, Part 1.* Addison Wesley, 1st edition. edition, 2011.

21 Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. Finding efficient circuits using SAT-solvers. In *SAT*, volume 5584 of *LNCS*, pages 32–44. Springer, 2009. doi:10.1007/978-3-642-02777-2_5.

22 Alexander S. Kulikov, Danila Pechenev, and Nikita Slezkin. SAT-based circuit local improvement. In *MFCS*, volume 241 of *LIPIcs*, pages 67:1–67:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.MFCS.2022.67.

23 Tung-Yuan Lee, Chia-Cheng Wu, Chia-Chun Lin, Yung-Chih Chen, and Chun-Yao Wang. Logic optimization with considering Boolean relations. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 761–766. IEEE, 2018. doi:10.23919/DATE.2018.8342109.

24 Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. A SAT approach to branchwidth. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *LNCS*, pages 179–195. Springer, 2016. doi:10.1007/978-3-319-40970-2_12.

25 Alan Mishchenko and Robert K. Brayton. SAT-based complete don't-care computation for network optimization. In *DATE*, pages 412–417. IEEE Computer Society, 2005.

26 Alan Mishchenko, Robert K. Brayton, Jie-Hong R. Jiang, and Stephen Jang. Scalable don't-care-based logic optimization and resynthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):34:1–34:23, 2011. doi:10.1145/2068716.2068720.

27 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Turbocharging treewidth-bounded Bayesian network structure learning. In *Proceeding of AAAI-21, the Thirty-Fifth AAAI Conference on Artificial Intelligence*, pages 3895–3903. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16508.

28 Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004. doi:10.1007/978-3-540-24730-2_3.

29 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. eSLIM. Software, This work was supported by the Vienna Science and Technology Fund (WWTF) under grant 10.47379/ICT19060, and the Austrian Science Fund (FWF) under grant 10.55776/W1255., swhId: swh:1:dir:9d2862aedfe458cfb06d4d43da6c0f46ed578c91 (visited on 2024-08-05). URL: https://github.com/fxreichl/eSLIM.

30 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Circuit minimization with QBF-based exact synthesis. In *AAAI*. AAAI Press, 2023. doi:10.1609/AAAI.V37I4.25524.

31 Heinz Riener, Winston Haaswijk, Alan Mishchenko, Giovanni De Micheli, and Mathias Soeken. On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis. In *DATE*, pages 1649–1654. IEEE, 2019. doi:10.23919/DATE.2019.8715185.

32 Heinz Riener, Alan Mishchenko, and Mathias Soeken. Exact dag-aware rewriting. In *DATE*, pages 732–737. IEEE, 2020. doi:10.23919/DATE48585.2020.9116379.

33 Hamid Savoj and Robert K. Brayton. The use of observability and external don't cares for the simplification of multi-level networks. In *DAC*, pages 297–301. IEEE Computer Society Press, 1990.

34 Andre Schidler and Stefan Szeider. SAT-boosted tabu search for coloring massive graphs. *J. Exp. Algorithmics*, 2825(1.5):1–19, 2023. doi:10.1145/3603112.

35 André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. *J. Artificial Intelligence Research*, 2024. To appear.

36 André Schidler and Stefan Szeider. Structure-guided local improvement for maximum satisfiability. In Peter Shaw, editor, *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Catalonia*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. To appear.

37 Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified Boolean formulas. In *ICTAI*, pages 78–84. IEEE, 2019.

38 Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *CP*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005. doi:10.1007/11564751_73.

39 Eleonora Testa, Luca G. Amarù, Mathias Soeken, Alan Mishchenko, Patrick Vuillod, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. Extending Boolean methods for scalable logic synthesis. *IEEE Access*, 8:226828–226844, 2020. doi:10.1109/ACCESS.2020.3045014.

40 University of California Berkeley. Berkeley logic interchange format (BLIF), 1992. URL: https://course.ece.cmu.edu/~ee760/760docs/blif.pdf.

41 Robert Wille, Hoang Minh Le, Gerhard W. Dueck, and Daniel Große. Quantified synthesis of reversible logic. In *DATE*, pages 1015–1020. ACM, 2008. doi:10.1109/DATE.2008.4484814.

Hierarchical Stochastic SAT and Quality Assessment of Logic Locking

Christoph Scholl
Department of Computer Science, University of Freiburg, Germany

Tobias Seufert
Department of Computer Science, University of Freiburg, Germany

Fabian Siegwolf
Department of Computer Science, University of Freiburg, Germany

Abstract

Motivated by the application of quality assessment of logic locking we introduce Hierarchical Stochastic SAT (HSSAT) which generalizes Stochastic SAT (SSAT). We look into the complexity of HSSAT and for solving HSSAT formulas we provide a prototype solver which computes exact evaluation results (i.e., without any approximation and without any imprecision caused by numerical rounding errors). Finally, we perform an intensive experimental evaluation of our HSSAT solver in the context of quality assessment of logic locking.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Automated reasoning

Keywords and phrases Stochastic Boolean Satisfiability, Hierarchical Stochastic SAT, Binary Decision Diagrams, Decision Procedure

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.24

Supplementary Material *Collection (Source Code and Benchmarks)*: `https://nc.informatik.uni-freiburg.de/index.php/s/PQ9zBAejC2ERCTD` [25]

Funding This work was supported by the German Research Foundation (DFG) within the project SUPRA (SCHO 894/7-1).

1 Introduction

Introducing Hierarchical Stochastic SAT (HSSAT) in this paper is motivated by the *logic locking* technique. Logic locking has been proposed to protect Integrated Circuits (ICs) from unauthorized usage. Such protection techniques have become necessary as the globalization of manufacturing of ICs may lead to trust issues between the various parties involved in the manufacturing process. Logic locking is a technique to prevent counterfeiting and overproduction by an untrusted foundry. Logic locking introduces additional logic to the IC which is connected to gates in the original IC as well as to a set of newly introduced inputs, the so-called *key inputs*, see Fig. 1 for an overview. The key inputs are stored in a tamper-proof memory and – at least ideally – the modified IC (i.e. "locked" IC) produces correct outputs only if the key inputs are set correctly. The correct key is not revealed to the foundry. The foundry just manufactures the ICs according to the mask data provided by the design house and delivers them to the design house. The design house loads a tamper-proof memory which is connected to the key inputs with the correct key value, thus it "activates" or "unlocks" the IC and delivers the activated IC to the end-users [31].

Numerous methods have been proposed for implementing logic locking schemes, see e.g. [23, 1, 21, 9, 22, 38, 37, 36, 40, 39]. Attacks against logic locking assume both an attacker model and an attack model. For the attacker, we assume that the design house is trusted, but the foundry and the end-user may be untrusted. This includes that the design house

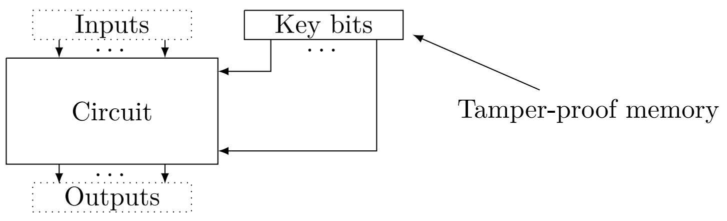


Figure 1 Logic locking.

keeps the unlocking key secret. If the workflow involves transferring the secret key, then the key is secured using cryptographic methods [23, 7] which is an orthogonal problem and is not considered in this paper. For the attack model, most attacks described in the literature assume that the locked netlist is known to the attacker and that the attacker either also has access to the original netlist [23] or at least to an unlocked device [21, 31, 38]. However, an untrusted foundry usually has only access to the mask data and not to the locked *netlist*. Thus, for such an attack it has to re-build the netlist from the mask data which is difficult and very expensive. Reverse engineering of the netlist based on a physical device is even harder or almost impossible for complex designs with small feature sizes and a non-trivial number of design layers. Therefore we assume here that the attacker does not have access to the locked netlist and we ask for the security of logic locking techniques against attacks without knowing the locked netlist. (Nevertheless, our attack model may assume that the attacker has an unlocked working device to check whether key guesses for the locked device have been successful.) Possible weaknesses of logic locking techniques could be: The IC may be unlocked not only with the original (intended) key, but also with other keys, or there may be many keys which "almost unlock" the IC which means that the IC produces for "almost all" input combinations the correct output combinations such that the IC can be used in practice with incorrect keys as well. This danger is real, since current logic locking methods like the one proposed by Yasin et al. [37] achieve resistance against SAT-based attacks [31] by keeping the fraction of input combinations with erroneous output combinations intentionally minimal for each incorrect key.

The goal of this paper is to provide formally precise methods for the quality assessment of logic locking without resorting to imprecise estimations based on simulation like in [19]. This is achieved by reducing the problem to known formalisms like Quantified Boolean Formulas (QBF), (Weighted) Model Counting, Projected (Weighted) Model Counting or Stochastic SAT (SSAT). However it turns out that answering certain interesting questions for quality assessment cannot be expressed naturally by those existing formalisms. For this, we introduce *Hierarchical* Stochastic SAT (HSSAT) which generalizes Stochastic SAT (SSAT). Note that a similar formalism called SSAT(Θ) has been developed by Fan and Jiang [11] in parallel to and independently from our work. Their motivation did not come from an application such as the quality assessment of logic locking, but from the theoretical question of generalizing counting formulas (CFs) that characterize the Counting Hierarchy [34] (as QBFs characterize the Polynomial Hierarchy [30]). We define syntax and semantics of HSSAT, we show that it is PSPACE complete (like QBF and SSAT), and we provide a prototype solver HSSATSolve for HSSAT which is based on ROBDDs [5] and is both algorithmically and numerically exact. In our experimental evaluation we apply our solver to several quality assessment problems. In cases where only (Weighted) Model Counting, Projected (Weighted) Model Counting, or SSAT is needed, we compared it to existing solvers like SharpSAT-TD [15], d4 [17], gpmc [32], arjun-ganak [26, 29], DC-SSAT [18], ClauSSat [6], ElimSSAT [35], and SharpSSAT [10]. The experiments show that HSSATSOLVE is capable of answering interesting questions that arise in quality assessment of logic locking and could not be mapped to already existing formalisms; for other questions it often outperforms existing solvers in our application domain.

The paper is structured as follows: We start with basic notations in Sect. 2. In Sect. 3 we introduce details on logic locking, quality assessment of logic locking and mapping it to known problems, whereas in Sect. 4 we discuss Hierarchical Stochastic SAT (HSSAT). We present experimental results in Sect. 5 and conclude with a summary and future research directions in Sect. 6.

2 Preliminaries and Notations

As usual a *Boolean formula* over the variable set X is either 0, 1, some variable $x \in X$, the negation $\neg\phi$ of a Boolean formula ϕ over X, the conjunction $\phi_1 \wedge \phi_2$, or the disjunction $\phi_1 \vee \phi_2$ of two Boolean formulas ϕ_1 and ϕ_2 over X. A *literal* is a variable or its negation. A *clause* is a disjunction of literals, a *cube* is a conjunction of literals. A conjunctive normal form (CNF) is a conjunction of clauses. An *assignment* α over $X' \subseteq X$ is a mapping from X' to $\mathbb{B} = \{0, 1\}$ which is called *partial*, if $X' \subset X$, and *full*, if $X' = X$. Sometimes we represent an assignment like $\alpha(x) = 0, \alpha(y) = 1$ by the corresponding cube $\neg xy$. The set of all assignments over X is denoted as $[\![X]\!]$. Given a formula ϕ and an assignment α over X, let $\phi[\alpha]$ denote the result of substituting every occurrence of variables $x \in X$ in ϕ with $\alpha(x)$ and evaluating the obtained boolean expression using the standard rules for the operators $\neg$, $\wedge$, and $\vee$. The Boolean formula ϕ represents a *Boolean function* $f_\phi : [\![X]\!] \to \mathbb{B}$ by $f_\phi(\alpha) = \phi[\alpha]$ for $\alpha \in [\![X]\!]$. 0 (resp. 1) represents the constant 0 (resp. 1) function. The *cofactor* $\phi|_\alpha$ of a Boolean formula ϕ wrt. some (full or partial) assignment α over $X' \subseteq X$ is a formula resulting from substituting every $x \in X'$ in ϕ with $\alpha(x)$.

Boolean functions can also be represented by *Reduced Ordered Binary Decision Diagrams (ROBDDs)* [5], see Fig. 5 for an example of an ROBDD. An ROBDD R over a set X of variables is a directed, acyclic graph $G = (V, E)$ having exactly one root $R.\texttt{root}$ with the following properties: V consists of terminal nodes (represented by squares in Fig. 5) and non-terminal (decision) nodes (represented by circles). The set of terminal nodes is a non-empty subset of $\{\boxed{0}, \boxed{1}\}$. The remaining non-terminal nodes $n \in V$ are labeled with variables $n.\texttt{var} := x \in X$ and they have exactly two outgoing edges, whose targets are denoted by $n.\texttt{low}$ resp. $n.\texttt{high} \in V$. The edge to $n.\texttt{low}$ is called the low edge (represented by a dashed line in Fig. 5), the edge to $n.\texttt{high}$ is called the high edge (represented by a solid line in Fig. 5). An ROBDD R is *ordered* which means that there is a global order $\pi : \{1, \ldots, |X|\} \to X$ such that on each path from the root $R.\texttt{root}$ to a terminal the variable labels of the non-terminal nodes occur in the order $\pi(1), \ldots, \pi(n)$. An ROBDD is *reduced*, i.e., it satisfies the following conditions: There is no pair of non-terminal nodes $n \neq m \in V$ with $n.\texttt{var} = m.\texttt{var}$, $n.\texttt{low} = m.\texttt{low}$, and $n.\texttt{high} = m.\texttt{high}$ ("isomorphism reduction"). For each non-terminal node $n \in V$ it holds $n.\texttt{low} \neq n.\texttt{high}$ ("Shannon reduction"). The reduction property makes ROBDD representations more compact and together with the ordering property it turns ROBDDs into canonical representations for given Boolean functions [5]. The Boolean function $\texttt{eval}(R.\texttt{root}) : [\![X]\!] \to \mathbb{B}$ defined by an ROBDD R can be computed recursively: $\texttt{eval}(\boxed{0}) = 0$, $\texttt{eval}(\boxed{1}) = 1$, and $\texttt{eval}(n) = (\neg n.\texttt{var} \wedge \texttt{eval}(n.\texttt{low})) \vee (n.\texttt{var} \wedge \texttt{eval}(n.\texttt{high}))$ for a non-terminal node n. It easily follows from this rule that the function value of $\texttt{eval}(R.\texttt{root})$ for a full assignment α can be computed by following a path through the ROBDD starting from $R.\texttt{root}$: At each node n on the path one follows the edge to $n.\texttt{low}$ iff $\alpha(n.\texttt{var}) = 0$ and the edge to $n.\texttt{high}$ iff $\alpha(n.\texttt{var}) = 1$. The function value of $\texttt{eval}(R.\texttt{root})$ for assignment α is then given by the reached terminal.

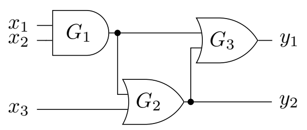


Figure 2 Original circuit.

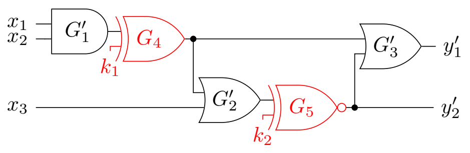


Figure 3 Locked circuit.

A *Stochastic Boolean Satisfiability* (SSAT) formula Φ in prenex form over variable set X is expressed by

$$\Phi = Q_1 x_1, \dots, Q_n x_n : \phi, \tag{1}$$

where ϕ is a Boolean formula over $X = \{x_1, \dots, x_n\}$, for $1 \leq i \leq n$ Q_i is either an *existential* quantifier $\exists$ or a *random* quantifier Я^{p_i} with $p_i \in [0,1]$. $Q_1 x_1, \dots, Q_n x_n$ is called the *prefix* and ϕ is called the *matrix* of Φ. The random quantifier Я^{p_i} on variable x_i indicates that $x_i = 1$ with probability $p_i \in [0,1]$ (resp. $x_i = 0$ with probability $1 - p_i$). The *semantics* of an SSAT formula Φ in the *optimization version* is a satisfying probability $\mathsf{Pr}[\Phi]$ recursively computed as follows:

1. $\mathsf{Pr}[\Phi] = 0$, if the matrix of Φ represents the 0 function,
2. $\mathsf{Pr}[\Phi] = 1$, if the matrix of Φ represents the 1 function,
3. $\mathsf{Pr}[\Phi] = \max(\mathsf{Pr}[\Phi'|_{\neg x_i}], \mathsf{Pr}[\Phi'|_{x_i}])$, if $\Phi = \exists x_i \Phi'$,
4. $\mathsf{Pr}[\Phi] = (1 - p_i) \cdot \mathsf{Pr}[\Phi'|_{\neg x_i}] + p_i \cdot \mathsf{Pr}[\Phi'|_{x_i}]$, if $\Phi = \text{Я}^{p_i} x_i \Phi'$.

Note that by the notion $\Phi'|_\alpha$ above we mean the SSAT formula where the prefix of Φ' is unchanged and its matrix ϕ' is replaced by the cofactor $\phi'|_\alpha$.

The *decision version* of an SSAT formula $\Phi = Q_1 x_1, \dots, Q_n x_n : \phi$ has the form Φ `op` q with `op` $\in \{<, \leq, >, \geq, =, \neq\}$, $q \in [0,1]$ and it evaluates to 1 (true), if $\mathsf{Pr}[\Phi]$ `op` q holds, and to 0 (false) otherwise.

If all quantifiers are random quantifiers, then SSAT corresponds to (weighted) model counting with $\texttt{weight}(x_i) = p_i$, $\texttt{weight}(\neg x_i) = 1 - p_i$.

The syntax of Quantified Boolean Formulas (QBFs) is as given by Eqn. (1) with the difference that in contrast to SSAT formulas Q_i is either an existential quantifier $\exists$ or a *universal* quantifier $\forall$. For the semantics of QBF, the item **4.** from the semantics definition above has to be replaced by $\mathsf{Pr}[\Phi] = \min(\mathsf{Pr}[\Phi'|_{\neg x_i}], \mathsf{Pr}[\Phi'|_{x_i}])$, if $\Phi = \forall x_i \Phi'$.

3 Logic Locking

As already mentioned in Sect. 1, logic locking changes an existing netlist by adding extra gates and extra key inputs, see Fig. 1. The circuit should only work correctly, if the correct key is applied to the key inputs. In the literature numerous logic locking schemes have been presented, see [23, 1, 21, 9, 22, 38, 37, 36, 40, 39], e.g.. The logic locking methods either modify the whole IC or only critical modules. Here we consider logic locking for combinational circuits without memory elements. For sequential circuits, we assume that logic locking modifies the combinational part.

▶ **Example 1.** Let us consider the very simple example of an original (non-locked) circuit in Fig. 2. Here we assume a simple logic locking method which randomly selects internal signals of the circuit and modifies them with logic locking. In the example the outputs of G_1 and G_2 are selected. An exor gate with key input k_1 is inserted at the output of G_1 and an exnor gate with key input k_2 is inserted at the output of G_2, see Fig. 3. It is easy to see that the key value $(k_1, k_2) = (0, 1)$ unlocks the circuit, since the Boolean functions implemented

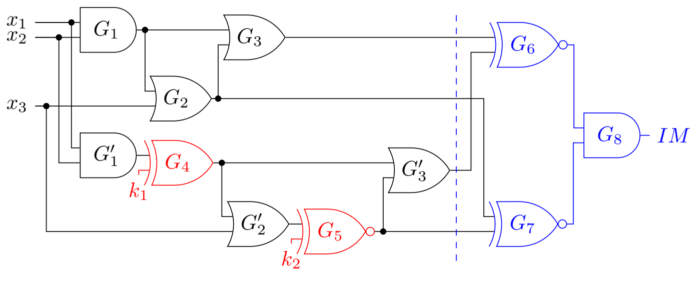


Figure 4 Inverted miter circuit.

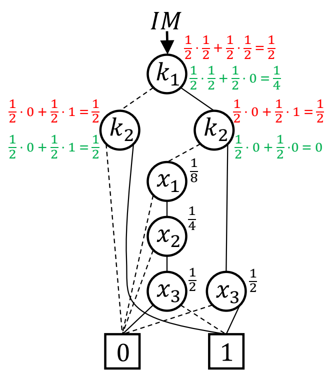

IM
$\frac{1}{2}\cdot\frac{1}{2}+\frac{1}{2}\cdot\frac{1}{2}=\frac{1}{2}$
k_1
$\frac{1}{2}\cdot\frac{1}{2}+\frac{1}{2}\cdot 0=\frac{1}{4}$
$\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 1=\frac{1}{2}$
k_2
$\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 1=\frac{1}{2}$
k_2
$\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 1=\frac{1}{2}$
$\frac{1}{2}\cdot 0+\frac{1}{2}\cdot 0=0$
x_1 $\frac{1}{8}$
x_2 $\frac{1}{4}$
x_3 $\frac{1}{2}$
x_3 $\frac{1}{2}$
0
1

Figure 5 ROBDD for inverted miter circuit.

by the original circuit and the locked circuit with $(k_1, k_2) = (0, 1)$ are the same. In contrast, $(k_1, k_2) = (1, 0)$ does not unlock the circuit, since e.g. for input $(x_1, x_2, x_3) = (1, 1, 1)$ the output of the original circuit is $(y_1, y_2) = (1, 1)$, but the output of the locked circuit is $(y'_1, y'_2) = (0, 0)$.

Now we are interested in a quality assessment of logic lockings for combinational circuits. For such a quality assessment we consider an inverted miter circuit for the original and the locked circuit. As usual, an inverted miter circuit connects the primary inputs of two circuits (in our case of the original and the locked circuit), connects the pairs of corresponding primary outputs of the two circuits with exnor (equivalence) gates, and connects the outputs of all exnor gates at the primary outputs with an and gate. I.e., the inverted miter circuit between the original and the locked circuit outputs a 1 for some assignment to the primary inputs (and key inputs) iff the corresponding outputs of the original and the locked circuit are identical. Fig. 4 shows the inverted miter circuit for the circuits of Figs. 2 and 3. Note that an assessment of logic locking quality is done by the (trusted) design house which owns both the original and the locked netlist. In this section we reduce several quality metrics to problems which are known in the formal reasoning community. The last metric then motivates the definition of a new problem called Hierarchical SSAT.

Key Existence

A very basic question is the question of key existence. If f_{IM} is the Boolean function at the output of the inverted miter, $\vec{X}$ is the input vector and $\vec{K}$ the key bit vector, then there exists an unlocking key iff the QBF

$$\exists \vec{K} \forall \vec{X} : f_{IM} \tag{2}$$

is satisfied (as already observed in [23]). Of course, the QBF should be satisfied, if logic locking was done correctly.

Key Uniqueness

The key existence does not imply security of logic locking however. It could be the case that the existing unlocking key is not the only unlocking key, but there are several unlocking keys. Therefore the uniqueness of the unlocking key is of interest as well. Let $\vec{K}_{\texttt{orig}}$ be the original (intended) unlocking key. Key uniqueness can simply be reduced to the QBF

$$\exists \vec{K} \forall \vec{X} : (f_{IM} \wedge (\vec{K} \neq \vec{K}_{\texttt{orig}})). \tag{3}$$

Fraction of Unlocking Keys

We cannot assume that logic locking methods always produce unique keys. Nevertheless, they may be appropriate in practice, if the number of unlocking keys is not too high. In this case the probability of guessing an unlocking key would be so low that an attack that randomly guesses keys will not have a high chance of success, especially if the size $|\vec{K}|$ is large enough. Thus, we are interested in the fraction of keys which are unlocking. This question can easily be reduced to the formula

$$\text{Ⅎ}^{0.5}\vec{K}\forall\vec{X} : f_{IM}. \tag{4}$$

Since existing SSAT solvers often do not support universal quantification, we have to transform Eqn. (4) into a version which removes $\forall$-quantification. This is not a trivial task for arbitrary SSAT formulas containing both $\exists$- and $\forall$-quantifiers, but in our special case it is easy. It is clear that a given fixed key $\vec{K}_{\texttt{fix}}$ *does not* unlock the locked circuit iff there is an input assignment which produces a 1 at the output of the *non-inverted* miter circuit, i.e., iff $\exists\vec{X}\neg f_{IM}|_{\vec{K}=\vec{K}_{\texttt{fix}}}$ holds. Thus, the *fraction* of keys which *do not* unlock the circuit is given by

$$\text{Ⅎ}^{0.5}\vec{K}\exists\vec{X} : \neg f_{IM}. \tag{5}$$

If this fraction is given by $p_{\texttt{w}}$, then the fraction of unlocking keys is $p_{\texttt{c}} = 1 - p_{\texttt{w}}$. After multiplying the result by $2^{|\vec{K}|}$ we obtain the number of unlocking keys. If the original key is unlocking and we are interested in the fraction of unlocking keys among the remaining keys, then we simply have to correct $p_{\texttt{c}}$ by computing $\frac{p_{\texttt{c}}\cdot 2^{|\vec{K}|}-1}{2^{|\vec{K}|}-1}$.

Existence of Keys with High Criticality

The existence of (many) keys different from the original key which completely unlock the circuit is of course a security issue. However, from an application perspective it is also critical, if there is a key which is different from the original key and "almost" unlocks the circuit. This is captured by the notion of "criticality" of a key.

▶ **Definition 2.** *The* criticality *of a key is defined as the quotient of the number of input assignments for which the key produces a correct output and the total number of input assignments.*

Given a criticality bound $c \in [0, 1]$, we are interested in the question whether there exists a key different from the original key $\vec{K}_{\texttt{orig}}$ with criticality $> c$. Keys with an extremely high criticality are considered to be dangerous, because with such a key an IC may possibly be operated for a long time without observing an error. Note that there are (as already mentioned in Sect. 1) logic locking methods [37] tailored towards resistance against SAT-based attacks [31] (which make the assumption that the locked netlist is at the attacker's disposal) where all keys have almost maximal criticality. This is of course undesirable from a practical point of view and therefore we consider it to be necessary to include criticality as a metric for assessing the quality of logic locking. For a fixed key $\vec{K}_{\texttt{fix}}$, $\text{Ⅎ}^{0.5}\vec{X} : f_{IM}|_{\vec{K}=\vec{K}_{\texttt{fix}}}$ computes the criticality of $\vec{K}_{\texttt{fix}}$ and thus the existence of a key (different from the original one) with criticality $> c$ can be checked by the SSAT formula

$$\Big(\exists\vec{K}\text{Ⅎ}^{0.5}\vec{X} : (f_{IM} \wedge (\vec{K} \neq \vec{K}_{\texttt{orig}}))\Big) > c. \tag{6}$$

According to the semantics definition reviewed in Sect. 2 the existential quantifier $\exists\vec{K}$ performs a maximization over all criticality values $\reflectbox{R}^{0.5}\vec{X} : f_{IM}|_{\vec{K}=\vec{K}_{\texttt{fix}}}$ for different fixed keys $\vec{K}_{\texttt{fix}}$ (the evaluation for the original key is forced to 0 by the additional constraint "$\vec{K} \neq \vec{K}_{\texttt{orig}}$"). This maximal value is finally compared to the criticality bound c.

Average Criticality of Keys

Again, it is not very dangerous, if there are only a few keys with high criticality, since the probability is not high that an attacker guesses a critical key. So we are also interested in the *average criticality* of all possible keys. This can be computed by the SSAT formula

$$\reflectbox{R}^{0.5}\vec{K}\reflectbox{R}^{0.5}\vec{X} : f_{IM}, \tag{7}$$

since the quantification $\reflectbox{R}^{0.5}\vec{K}$ just averages over all possible criticalities of the key values. Eqn. (7) may be interpreted as a model counting problem.

Fraction of Keys with High Criticality

A high average criticality may be regarded as a security problem at first sight. Unfortunately, it is not a perfect metric for assessing the quality of logic locking, since there may be several reasons for a high average criticality. Let us look into two extreme cases: In case 1 all keys (except the original one) have criticality 0.5. This leads to a (rather high) average criticality of (about) 0.5 (the original key contributes 1, all other keys 0.5 to the average). In case 2, one half of the keys have criticality 1, the other half have criticality 0. Here the average criticality is again 0.5. Case 1 is not really critical, since the user most probably cannot work with a key with criticality 0.5, since half of the input assignments produce erroneous values. The probability of guessing a key with criticality > 0.5 is as low as possible ($\frac{1}{2^{|\vec{K}|}}$). On the other hand, case 2 with almost the same average criticality is highly critical from an application point of view. On average, every second guessed key completely unlocks the circuit.

This observation shows that we should look for a different criticality-based metric. We would like to ensure that the *number* of keys (or the *fraction* of keys) with a criticality larger than a given criticality bound $c \in [0,1]$ is as low as possible. If we choose the criticality bound c in a way that the circuit is supposed to be of no use, if it is operated with a key with criticality $\leq c$, then this fraction immediately gives the probability of guessing a key which can be used in practice. From a conceptional point of view, the task is to compute for each fixed key $\vec{K}_{\texttt{fix}}$ the probability $\reflectbox{R}^{0.5}\vec{X} : f_{IM}|_{\vec{K}=\vec{K}_{\texttt{fix}}}$ (i.e. the criticality of $\vec{K}_{\texttt{fix}}$), evaluate "$\reflectbox{R}^{0.5}\vec{X} : f_{IM}|_{\vec{K}=\vec{K}_{\texttt{fix}}} > c$" for the given criticality bound c, and then just to compute the fraction of keys for which this evaluation returns "true". In the end we have to compare this fraction again with a bound which specifies how large the computed fraction is allowed to be. However, this approach does not seem to be feasible, since it leads to a number of SSAT (or model counting) problems which is exponential in the number of key bits $\vec{K}$. The problem cannot naturally be reduced to an SSAT formula, since the comparison with the criticality bound c has to be performed for each key value individually instead of a single comparison as in Eqn. (6). This situation leads us to the definition of a new generalization of SSAT called *Hierarchical* Stochastic SAT (HSSAT). HSSAT allows to hierarchically include several comparison operators within the formula instead of only one at the end as in the decision version of SSAT.

4 Hierarchical Stochastic SAT

4.1 Definition of HSSAT

Here we define our new formalism of HSSAT which has been motivated in the previous section. In contrast to usual definitions of SSAT we also allow universal quantifiers in HSSAT, since (1) allowing them does not change the complexity class of HSSAT and (2) we can easily process universal quantifiers in our solver as well, so there is no reason to forbid universal quantifiers.

▶ **Definition 3** (Syntax of HSSAT formulas). *A Boolean Formula ϕ over variable set X is also an HSSAT formula Φ with matrix $\mathtt{matrix}(\Phi) = \phi$, the set $\mathtt{FV}(\Phi) = X$ of free variables, the set $\mathtt{BV}(\Phi) = \emptyset$ of bound variables, the quantifier order $\pi(\Phi) : \{1, \dots, |\mathtt{BV}(\Phi)|\} \to \mathtt{BV}(\Phi)$, and hierarchy level $\mathtt{hlevel}(\Phi) = 0$.*

Now let Φ be an arbitrary HSSAT formula with matrix $\mathtt{matrix}(\Phi)$, the set $\mathtt{FV}(\Phi)$ of free variables, the set $\mathtt{BV}(\Phi)$ of bound variables, the quantifier order $\pi(\Phi) : \{1, \dots, |BV(\Phi)|\} \to BV(\Phi)$, and hierarchy level $\mathtt{hlevel}(\Phi)$. Let $x \in \mathtt{FV}(\Phi)$, $p, q \in [0, 1]$, $\mathtt{op} \in \{<, \leq, >, \geq, =, \neq\}$. Then the following formulas are HSSAT formulas as well:

(a) *$\Phi' = (\exists x \Phi)$ is an HSSAT formula with $\mathtt{FV}(\Phi') = \mathtt{FV}(\Phi) \setminus \{x\}$, $\mathtt{BV}(\Phi') = \mathtt{BV}(\Phi) \cup \{x\}$, $\mathtt{hlevel}(\Phi') = \mathtt{hlevel}(\Phi)$, $\mathtt{hlevel}(x) = \mathtt{hlevel}(\Phi)$, $\mathtt{quantor}(x) = \exists$.*

(b) *$\Phi' = (\forall x \Phi)$ is an HSSAT formula with $\mathtt{FV}(\Phi') = \mathtt{FV}(\Phi) \setminus \{x\}$, $\mathtt{BV}(\Phi') = \mathtt{BV}(\Phi) \cup \{x\}$, $\mathtt{hlevel}(\Phi') = \mathtt{hlevel}(\Phi)$, $\mathtt{hlevel}(x) = \mathtt{hlevel}(\Phi)$, $\mathtt{quantor}(x) = \forall$.*

(c) *$\Phi' = (\rotatebox[origin=c]{180}{R}^p x \Phi)$ is an HSSAT formula with $\mathtt{FV}(\Phi') = \mathtt{FV}(\Phi) \setminus \{x\}$, $\mathtt{BV}(\Phi') = \mathtt{BV}(\Phi) \cup \{x\}$, $\mathtt{hlevel}(\Phi') = \mathtt{hlevel}(\Phi)$, $\mathtt{hlevel}(x) = \mathtt{hlevel}(\Phi)$, $\mathtt{quantor}(x) = \rotatebox[origin=c]{180}{R}^p$.*

(d) *$\Phi' = (\Phi \ \mathtt{op} \ q)$ is an HSSAT formula with $\mathtt{FV}(\Phi') = \mathtt{FV}(\Phi)$, $\mathtt{BV}(\Phi') = \mathtt{BV}(\Phi)$, $\mathtt{hlevel}(\Phi') = \mathtt{hlevel}(\Phi) + 1$, $\mathtt{op}(\mathtt{hlevel}(\Phi) + 1) = \mathtt{op}$, $\mathtt{prob}(\mathtt{hlevel}(\Phi) + 1) = q$.*

In all cases (a) - (d) we have $\mathtt{matrix}(\Phi') = \mathtt{matrix}(\Phi)$. In cases (a) - (c) the quantifier order is $\pi(\Phi') : \{1, \dots, |BV(\Phi')|\} \to BV(\Phi')$ with $\pi(\Phi')(1) = x$, $\pi(\Phi')(i) = \pi(\Phi)(i-1)$ for all $2 \leq i \leq |BV(\Phi')|$. In case (d) $\pi(\Phi') = \pi(\Phi)$.

An HSSAT formula Φ is called closed, *if $\mathtt{FV}(\Phi) = \emptyset$.*

In a closed HSSAT formula all variables in the matrix are bound by $\exists$, $\forall$, or $\rotatebox[origin=c]{180}{R}$ quantifiers. We define the semantics only for closed HSSAT formulas and assume that non-closed formulas are made closed by using leading existential quantifiers.

Note that SAT formulas can be seen as closed HSSAT formulas of hierarchy level 0 with $\exists$-quantifiers only, QBFs as closed HSSAT formulas of hierarchy level 0 with only $\exists$- and $\forall$-quantifiers, SSAT formulas in the optimization version as closed HSSAT formulas of hierarchy level 0 with only $\exists$- and $\rotatebox[origin=c]{180}{R}$-quantifiers, and SSAT formulas in the decision version as closed HSSAT formulas of hierarchy level 1 with only $\exists$- and $\rotatebox[origin=c]{180}{R}$-quantifiers.

Before we look into the semantics of HSSAT, we consider how to express the motivating example from the last section with an HSSAT formula.

Fraction of Keys with High Criticality

The problem of deciding whether the fraction of keys with criticality $> c$, $c \in [0, 1]$ is larger than $d \in [0, 1]$ (see Sect. 3) can be reduced to the following closed HSSAT formula:

$$((\rotatebox[origin=c]{180}{R}^{0.5} \vec{K} ((\rotatebox[origin=c]{180}{R}^{0.5} \vec{X} : f_{IM}) > c)) > d). \tag{8}$$

▶ **Example 4.** For Example 1 (see Figs. 2, 3, 4) the corresponding closed HSSAT formula is

$$\Phi := \left[\left(\rotatebox[origin=c]{180}{R}^{0.5} k_1 \rotatebox[origin=c]{180}{R}^{0.5} k_2 \left[\left(\rotatebox[origin=c]{180}{R}^{0.5} x_1 \rotatebox[origin=c]{180}{R}^{0.5} x_2 \rotatebox[origin=c]{180}{R}^{0.5} x_3 : f_{IM} \right) > c \right] \right) > d \right].\,[1] \tag{9}$$

Φ is a closed formula with an empty set $\texttt{FV}(\Phi)$ of free variables, the bound variables are $\texttt{BV}(\Phi) = \{k_1, k_2, x_1, x_2, x_3\}$. For all variables $x \in \texttt{BV}(\Phi)$ $\texttt{quantor}(x) = \text{Я}^{0.5}$. The quantifier order is given by $\pi(\Phi)(1) = k_1$, $\pi(\Phi)(2) = k_2, \ldots, \pi(\Phi)(5) = x_3$. The hierarchy level $\texttt{hlevel}(\Phi)$ is 2, $\texttt{hlevel}(x_1) = \texttt{hlevel}(x_2) = \texttt{hlevel}(x_3) = 0$, $\texttt{hlevel}(k_1) = \texttt{hlevel}(k_2) = 1$. The probability value for comparison at hierarchy level 1 is $\texttt{prob}(1) = c$ and at hierarchy level 2 it is $\texttt{prob}(2) = d$. The operands at hierarchy levels 1 and 2 are $>$: $\texttt{op}(1) = \texttt{op}(2) = >$.

The formula checks whether the fraction of keys with criticality greater than c is greater than d.

The semantics of HSSAT formulas can be formally defined using an evaluation function $\textsc{Eval}(\cdot)$. The result is a value from $[0, 1]$ which may be interpreted as a probability value or – if the HSSAT formula is a comparison with some number from $[0, 1]$, i.e., if we are in case (d) of Def. 3 – as a logical value 0 (false) or 1 (true).

▶ **Definition 5** (Semantics of HSSAT formulas). *Let Φ' be a closed HSSAT formula. Φ' is evaluated by a function $\textsc{Eval}(\cdot)$.*

(1) *If $\texttt{BV}(\Phi') = \emptyset$, $\texttt{hlevel}(\Phi') = 0$, and $f_{\Phi'} = 0$, then $\textsc{Eval}(\Phi') = 0$.*
(2) *If $\texttt{BV}(\Phi') = \emptyset$, $\texttt{hlevel}(\Phi') = 0$, and $f_{\Phi'} = 1$, then $\textsc{Eval}(\Phi') = 1$.*
(3) *If $\Phi' = \exists x \Phi$, then $\textsc{Eval}(\Phi') = \max(\textsc{Eval}(\Phi|_{\neg x}), \textsc{Eval}(\Phi|_x))$.*
(4) *If $\Phi' = \forall x \Phi$, then $\textsc{Eval}(\Phi') = \min(\textsc{Eval}(\Phi|_{\neg x}), \textsc{Eval}(\Phi|_x))$.*
(5) *If $\Phi' = \text{Я}^p x \Phi$, then $\textsc{Eval}(\Phi') = (1 - p) \cdot \textsc{Eval}(\Phi|_{\neg x}) + p \cdot \textsc{Eval}(\Phi|_x)$.*
(6) *If $\Phi' = (\Phi \ \texttt{op} \ q)$, then $\textsc{Eval}(\Phi') = 1$ in case $\textsc{Eval}(\Phi) \ \texttt{op} \ q$ holds, $\textsc{Eval}(\Phi') = 0$ otherwise.*

The notion $\Phi|_\alpha$ above means that the matrix of Φ is replaced with its cofactor $(\texttt{matrix}(\Phi))|_\alpha$.

4.2 Complexity of HSSAT

Now we come to the complexity of the HSSAT problem. It is pretty easy to see that HSSAT is PSPACE complete – just as SSAT.

▶ **Lemma 6.** *HSSAT is PSPACE hard.*

Proof. Since we allow $\forall$-quantifiers in HSSAT, each QBF is a closed HSSAT formula of hierarchy level 0 with $\exists$- and $\forall$-quantifiers only. Thus the hardness proof easily follows from the PSPACE hardness of QBF. If $\forall$-quantifiers would not be allowed in HSSAT, a polynomial time reduction from QBF would replace in a QBF Φ all $\forall$-quantifiers by Я^p-quantifiers with arbitrary $p \in (0, 1)$, leading to some Φ', and it would consider "$\Phi' = 1$". The QBF Φ is satisfied iff $\Phi' = 1$ evaluates to 1 (true). (This is just as in the reduction for SSAT [18].) ◀

▶ **Lemma 7.** *HSSAT is in PSPACE.*

Proof (Sketch). Similar to the SSAT case, Def. 5 immediately suggests an evaluation of a closed HSSAT formula Φ with quantified variables from X by an "implicit depth-first traversal of the decision tree" of all assignments over X. The decision tree considers the variables in the order they occur in the prefix of Φ. Of course, it is not necessary to store the decision tree explicitly, but it is only necessary to store the currently considered path. Whenever a node has been evaluated, it is not necessary anymore to store the evaluation values for its successors. Thus, it is never needed to store more than $O(|X|)$ values during the depth-first traversal. Moreover, it is easy to see that the size of the number representations occurring during the evaluation is polynomially restricted by the input size of the problem instance. Thus, HSSAT is in PSPACE. ◀

[1] Parentheses according to Def. 3 which are clear from the context are omitted.

Algorithm 1 evalHSSAT.

```
Input: HSSAT formula $\Phi$ with $\texttt{matrix}(\Phi) = \texttt{ROBDD}\ R$, $\texttt{FV}(\Phi) = \emptyset$
Output: rational number evaluating $\Phi$
1: return $\textsc{evalEdge}(\textsc{evalNode}(R.\texttt{root}), \texttt{hlevel}(\Phi), \texttt{hlevel}(R.\texttt{root}))$;
```

Algorithm 2 evalNode.

```
Input: $\texttt{ROBDD}$ node $n$
Output: rational number evaluating $n$
1: if $n = \boxed{0}$ then return $0$;
2: if $n = \boxed{1}$ then return $1$;
3: if $n.\texttt{value} \neq \texttt{undefined}$ then return $n.\texttt{value}$;
4: $p_{\texttt{low}} := \textsc{evalEdge}(\textsc{evalNode}(n.\texttt{low}), \texttt{hlevel}(n), \texttt{hlevel}(n.\texttt{low}))$;
5: $p_{\texttt{high}} := \textsc{evalEdge}(\textsc{evalNode}(n.\texttt{high}), \texttt{hlevel}(n), \texttt{hlevel}(n.\texttt{high}))$;
6: if $\texttt{quantor}(n.\texttt{var}) = \exists$ then $p := \max(p_{\texttt{low}}, p_{\texttt{high}})$;
7: if $\texttt{quantor}(n.\texttt{var}) = \forall$ then $p := \min(p_{\texttt{low}}, p_{\texttt{high}})$;
8: if $\texttt{quantor}(n.\texttt{var}) = ᗡ^q$ then $p := (1-q) \cdot p_{\texttt{low}} + q \cdot p_{\texttt{high}}$;
9: $n.\texttt{value} := p$;
10: return $p$;
```

Lemma 6 and Lemma 7 imply Theorem 8.

▶ **Theorem 8.** *HSSAT is PSPACE complete.*

4.3 Solving HSSAT

Here we present a prototype algorithm for solving HSSAT which is based on the computation of ROBDDs for the matrix of the formula. The prototype algorithm also gives an indication of how to design a generalization of DPLL-based SSAT algorithms like DC-SSAT [18], Prime [24], or SharpSSAT [10] to HSSAT.

We will start with an algorithm which first builds for an HSSAT formula Φ an ROBDD $R = (V, E)$ for $\texttt{matrix}(\Phi)$ with the variable order $\pi := \pi(\Phi)$. Then the evaluation of Φ is reduced to an evaluation of the ROBDD R by Algs. 1, 2, and 3. Alg. 2 is a recursive algorithm computing the evaluation values at the different ROBDD nodes and Alg. 3 takes care of edges in the ROBDD, especially long edges crossing several levels in the variable order. Alg. 1 just reads out the correct value from the root of the ROBDD. To simplify notations in the definition of Algs. 1 and 2 we define for $n \in V$: $\texttt{hlevel}(n) = 0$, if $n \in \{\boxed{0}, \boxed{1}\}$, $\texttt{hlevel}(n) = \texttt{hlevel}(n.var)$ otherwise. We will consider the correctness of the algorithms first and we will show an example afterwards.

For the time being, we assume that the HSSAT formula does not contain hierarchical comparisons with values $p \in [0, 1]$, i.e., we first neglect case (d) of Def. 3. This means that the hierarchy level of Φ as well as of all variables is 0. It is easy to check that in this case Alg. 3 does not have any effect, i.e., it returns its input probability without any change.

I.e., with our initial assumption the algorithm works like an ROBDD-based algorithm for SSAT. We briefly discuss its correctness. First we assume that the "Shannon reductions are reverted" in the ROBDD. This means the following: As long as there exists a "long" edge from a node n to node $m = n.\texttt{dir}$, $\texttt{dir} \in \{\texttt{low}, \texttt{high}\}$, with either $m \in \{\boxed{0}, \boxed{1}\}$ and $\pi^{-1}(n.\texttt{var}) < |X|$ or $\pi^{-1}(m.\texttt{var}) > \pi^{-1}(n.\texttt{var}) + 1$, we introduce a new successor n' of n

Algorithm 3 EVALEDGE.

```
Input: rational number $p_{\texttt{target}}$ evaluating the target node of an edge, hierarchy level $hl_{\texttt{source}}$
    of source node, hierarchy level $hl_{\texttt{target}}$ of target node of the edge
Output: rational number evaluating the edge between source and target node
1: $p := p_{\texttt{target}}$; $hl := hl_{\texttt{target}}$;
2: while $hl < hl_{\texttt{source}}$ do
3:     if $p$ op$(hl + 1)$ prob$(hl + 1)$ then $p := 1$ else $p := 0$;
4:     $hl := hl + 1$;
5: return $p$;
```

with $n.\texttt{dir} = n'$, $n'.\texttt{low} = m$, $n'.\texttt{high} = m$. We perform a similar transformation for the root of R, if $\pi^{-1}(R.\texttt{root}) > 1$. The resulting ROBBD is then essentially a decision tree with variable order π (with the only difference that there are shared nodes in the ROBDD due to "isomorphism reduction"). Now it is clear that Alg. 2 does exactly the same evaluation steps as given in Def. 5, since following an edge from some node n labelled with $n.\texttt{var}$ in the ROBDD exactly corresponds to a cofactor computation (remember that Alg. 3 does not have an effect for now). Alg. 2 additionally caches values which have already been computed in the variable $n.\texttt{value}$. It only remains to show that re-introducing Shannon reductions does not change the evaluation. Consider a node n with $n.\texttt{low} = n.\texttt{high}$. From $v = \max(v, v) = \min(v, v) = (1 - p) \cdot v + p \cdot v$ it easily follows by case distinction with the cases $\texttt{quantor}(n.\texttt{var}) = \exists, \forall, Ⅎ^p$ that the evaluation of n gives the same value as the evaluation of $n.\texttt{low} = n.\texttt{high}$, i.e., removing n does not change the evaluation.

Now that it is clear that the algorithm is correct, if Φ does not contain hierarchical comparisons according to case (d) of Def. 3, we only have to consider what changes, if we re-introduce case (d). Intuitively, Rule (6) of Def. 5 with $\Phi' = (\Phi \texttt{ op } q)$ and hierarchy level $\texttt{hlevel}(\Phi') = hl$ says the following: If we are at a node n with hierarchy level $\texttt{hlevel}(n) = hl$ and read a value v from a node m with hierarchy level $\texttt{hlevel}(m) = hl - 1$, then we should not take v itself, but we have to replace v with the outcome of the comparison $v \texttt{ op } q$ which is 0 or 1. The **while** loop in Alg. 3 just accounts for the case that there may be "long edges" from a node n with $\texttt{hlevel}(n) = hl_n$ to node m with $\texttt{hlevel}(m) = hl_m$ and $hl_n \geq hl_m + 1$ in the ROBDD, crossing several hierarchy levels. Then several evaluations are needed, comparing with $\texttt{op}(hl_m + 1)$ $\texttt{prob}(hl_m + 1)$ first, and finally with $\texttt{op}(hl_n)$ $\texttt{prob}(hl_n)$. In a similar way, the call to EVALEDGE in Alg. 1 considers the case that $\texttt{hlevel}(\Phi) > \texttt{hlevel}(R.\texttt{root})$, i.e., the outer formula contains comparisons according to case (d) of Def. 3.

▶ **Theorem 9.** *Algorithm 1 correctly evaluates an HSSAT formula Φ according to Def. 5.*

▶ **Example 10.** Consider the ROBDD in Fig. 5 for the inverted miter circuit in Fig. 4. Let us consider the HSSAT formula $\left[\left(Ⅎ^{0.5}k_1Ⅎ^{0.5}k_2 \left[\left(Ⅎ^{0.5}x_1Ⅎ^{0.5}x_3Ⅎ^{0.5}x_3 : f_{IM}\right) \geq c\right]\right) \geq 0.3\right]$, see also Example 4. Let us first assume $c = \frac{1}{2}$. The nodes in the ROBDD are annotated with their evaluation values. The evaluation for the x_i-nodes just uses the formula $\frac{1}{2} \cdot$ EVALNODE$(n.\texttt{low}) + \frac{1}{2} \cdot$ EVALNODE$(n.\texttt{high})$. However, the evaluation of the right k_2-node, e.g., has to consider the comparison with $c = \frac{1}{2}$. For the $\texttt{low}$–successor we have $\frac{1}{8} < \frac{1}{2}$, thus we have to replace its value by 0. For the $\texttt{high}$–successor we have $\frac{1}{2} \geq \frac{1}{2}$, thus we have to replace its value by 1. This leads to $\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2}$ at the right k_2-node. The upper (red) annotations at the k_i-nodes give the node values for $c = \frac{1}{2}$ with the final value of $\frac{1}{2}$ at the root, leading to 1 in the final comparison with 0.3. The lower (green) annotations at the k_i-nodes give the node values for $c = 0.99$ with the final value of $\frac{1}{4}$ at the root, leading to 0 in the final comparison with 0.3.

Improvements

Here we consider two improvements to the basic algorithm.

The first improvement concerns the variable order in the ROBDD. For this improvement we define quantifier blocks for HSSAT, similar to SSAT or QBF. For an HSSAT formula Φ with bound variables $\texttt{BV}(\Phi) = X$, we define a partition $P = \{X_1, \ldots, X_k\}$ on X by the following rule: P is the (unique) partition where for all $1 \leq i \leq |X| - 1$ $\pi(\Phi)(i)$ and $\pi(\Phi)(i + 1)$ are in the same set X_j iff $\texttt{quantor}(\pi(\Phi)(i)) = \texttt{quantor}(\pi(\Phi)(i + 1))$ and $\texttt{hlevel}(\pi(\Phi)(i)) = \texttt{hlevel}(\pi(\Phi)(i + 1))$ (random quantors $\reflectbox{R}^p$ and $\reflectbox{R}^q$ are considered to be equal even if $p \neq q$). Thus P partitions the variables in the prefix into groups of consecutive variables with the same quantor and the additional condition that comparisons according to case (d) of Def. 3 introduce cuts into those groups. It is easy to see by a simple computation that the value of an HSSAT Φ does not change, if neighboring variables $\pi(\Phi)(i)$ and $\pi(\Phi)(i+1)$ belonging to the same group X_j are exchanged in Φ. Thus, all variables within some group X_j can be arbitrarily exchanged without changing the value of Φ. This also means that for the ROBDD-based evaluation we do not need to choose exactly the variable order $\pi(\Phi)$, but we can choose a variable order which is "*compatible with* $\pi(\Phi)$". "Compatible" means here that variables within groups X_j of $\{X_1, \ldots, X_k\}$ can be arbitrarily exchanged in the variable order of the ROBDD without affecting the value of the evaluation. In our implementation we choose $\pi(\Phi)$ as the initial order when we build the ROBDD for $\texttt{matrix}(\Phi)$, but we activate the dynamic reordering technique of "group sifting" [20] which dynamically tries to change the variable order within the blocks with the goal of minimizing the number of needed ROBDD nodes to represent $\texttt{matrix}(\Phi)$.

The second improvement comes into play when the matrix of the HSSAT formula Φ is given in CNF, especially when it was produced from a circuit representation by a transformation like Tseitin transformation [33]. In our implementation we use the "interpolation-based gate extraction" by Slivovsky [27] which is based on the replacement of so-called "defined variables" by their "definition".

▶ **Definition 11** ([27]). *Let ϕ be a Boolean formula over X, $X' \subseteq X$. $x \in X$ is* defined *in terms of X' in ϕ iff $\alpha(x) = \beta(x)$ for any two full assignments α and β that satisfy ϕ and agree on X'. A definition of x by X' in ϕ is a formula ψ over X' with $\alpha(x) = \psi[\alpha]$ for any assignment α that satisfies ϕ.*

Slivovsky [27] looks into QBF and considers existential defined variables and universal defining variables. We adjust the approach of [27] to HSSAT by the following lemma:

▶ **Lemma 12.** *Let x be an existential variable of a closed HSSAT formula Φ and for all hl with $1 \leq hl \leq \texttt{hlevel}(x)$ the equation "0 $\texttt{op}(hl)$ $\texttt{prob}(hl)$" does* not *hold. Let x be defined in terms of $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ in $\texttt{matrix}(\Phi)$ and let ψ be a corresponding definition. Then $\textsc{Eval}(\Phi) = \textsc{Eval}(\Phi[x := \psi])$ where $\Phi[x := \psi]$ results from Φ by replacing x with ψ in the matrix and omitting $\exists x$ in the prefix.*

On the one hand, Lemma 12 does not only allow universal variables to the left of existential variables as defining variables, but *all* variables to the left. (A similar approach has been already proposed in [27] as an improvement for the restricted case of QBFs.) On the other hand, we need an additional restriction on the choice of defined existential variables that is specific to *Hierarchical* Stochastic SAT. We prove the correctness of Lemma 12 in Appendix A.

The transformation of Lemma 12 is useful, since the replaced variables do not occur as input variables in the ROBDD later on and the computed ROBDDs are smaller in most cases. Semantic gate extraction has already been used in the context of SSAT solving [35]. Similar ideas have also been used in [16, 14] for model counting.

5 Experiments

We implemented our prototype solver HSSATSolve with the ROBDD package CUDD [28] using dynamic group sifting [20] for optimizing the variable orders. For all number representations we use rational numbers from the GNU Multiple Precision Arithmetic Library (GMP) [12]. All experiments were performed on one core of an Intel Xeon CPU E5-2650v2 CPU with 2.6 GHz using 16 GB of memory with a timeout of 30 CPU minutes. All benchmarks and the code of our tool can be found at [25].

For our experiments we added logic locking to the combinational ISCAS'85 benchmarks [4] and to the combinational parts of the ITC'99 benchmarks [8]. For logic locking we used the method from [23] which randomly selects signals in the circuit and replaces them randomly either with exor or exnor gates having a key bit as a side input (see also Figs. 2 and 3). We considered key bit lengths of 4, 8, 16, 32, and 64. We generated the inverted miters of the original and the locked circuits and generated HSSAT formulas of types ꓤ∃, ꓤ, ∃ꓤ, and H. Formulas of type ꓤ∃ compute the fraction of unlocking keys (Eqn. (5)), formulas of type ꓤ the average criticality of keys (Eqn. (7)), formulas of type ∃ꓤ the existence of keys with high criticality (Eqn. (6)), and formulas of type H the fraction of keys with high criticality (Eqn. (8)). For all random quantifiers we considered input probabilities of 0.5.

To be able to evaluate the general quality of our prototype solver we first compared it with existing tools. For this we translated the formulas of types ꓤ∃, ꓤ, and ∃ꓤ into CNFs by Tseitin transformation [33]. Formulas of type ꓤ are Model Counting problems, formulas of type ꓤ∃ are Projected Model Counting problems. For their comparisons, we considered the best exact solvers of the Model Counting Competition 2023 [13]. We used SharpSAT-TD [15], d4 [17], gpmc [32], and arjun-ganak [26, 29]. Formulas of type ∃ꓤ are SSAT formulas. Here we compared with DC-SSAT [18], ClauSSat [6], ElimSSAT [35], and SharpSSAT [10].

For formulas with matrix in CNF we use a version HSSATSolve$^{\texttt{cnf}}$ of our tool. Here we first apply the tool UNIQUE [27] which was adapted according to Lemma 12 to transform the CNF into a circuit format, then we optimize the circuit using ABC [2, 3], and finally we build ROBDDs for the matrix using CUDD with dynamic group sifting activated. Then we evaluate the ROBDDs using Alg. 1. For formulas with matrix in circuit format we use a version HSSATSolve$^{\texttt{circ}}$ of our tool where the application of UNIQUE is of course omitted.

Fig. 6 shows a cactus plot for the comparisons using formulas of type ꓤ∃, Fig. 7 a cactus plot for the comparisons using formulas of type ꓤ, and Fig. 8 a cactus plot for the comparisons using formulas of type ∃ꓤ.

Overall, it turns out that our prototype solver competes well with the already existing tools on formulas from our application domain of logic locking. Comparing HSSATSolve$^{\texttt{cnf}}$ and HSSATSolve$^{\texttt{circ}}$ the results show that HSSATSolve$^{\texttt{circ}}$ clearly outperforms HSSATSolve$^{\texttt{cnf}}$. The gap between HSSATSolve$^{\texttt{cnf}}$ and HSSATSolve$^{\texttt{circ}}$ proves that the semantic gate extraction procedure of UNIQUE is apparently not able to reconstruct the original circuit structure from the CNF – although for our benchmarks this is easily possible with a simple syntactic gate extraction method. Based on those observations we believe that it will be possible to extend UNIQUE with certain improvements, but this will be the subject of future work.

There are apparent differences between the results of Figs. 6 and 7, e.g.. Whereas gpmc and arjun-ganak solve more formulas than HSSATSolve$^{\texttt{circ}}$ on formulas of type ꓤ∃ (Fig. 6), the situation is the other way round for SharpSAT-td and d4 on formulas of type ꓤ (Fig. 7). In particular, HSSATSolve$^{\texttt{circ}}$ solves more formulas of type ꓤ than formulas of type ꓤ∃. To analyze the reason for this, we added an experiment with HSSATSolve$^{\texttt{circ,ns}}$ which is a

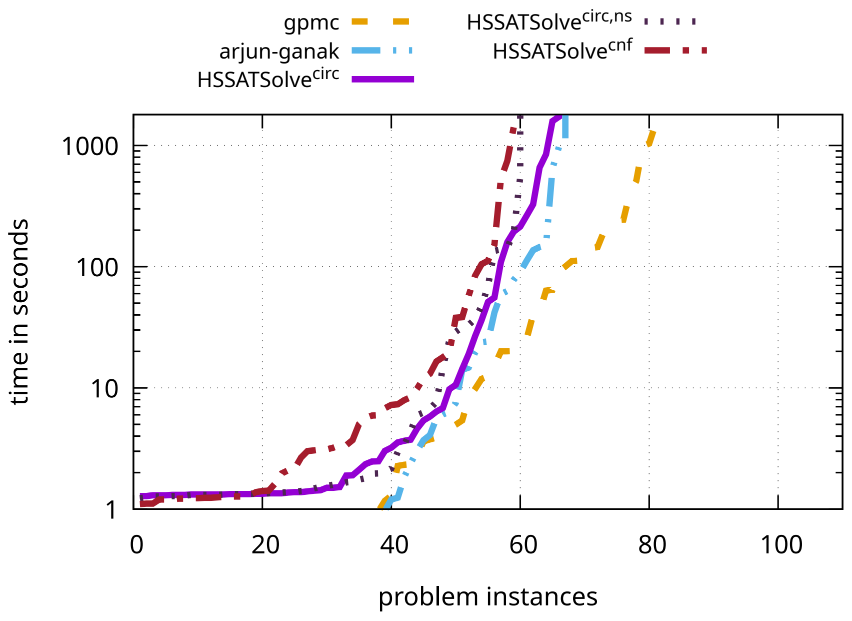


Figure 6 Type ∃Я.

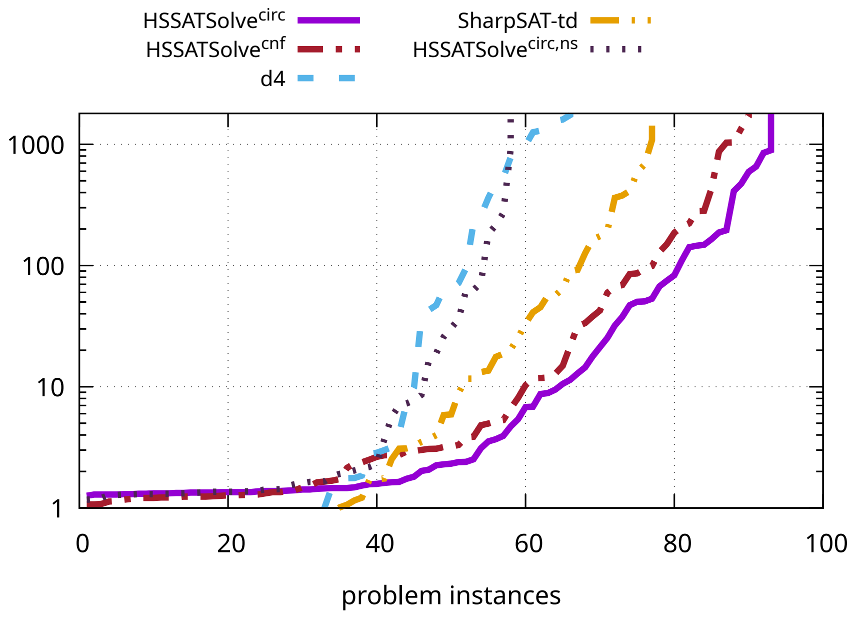


Figure 7 Type Я.

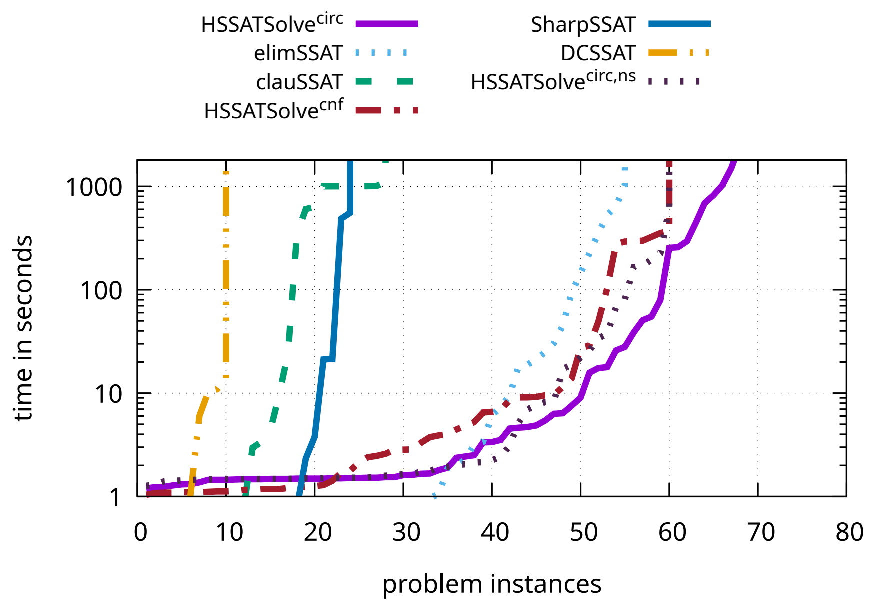


Figure 8 Type ∃Ɐ.

version of HSSATSolve$^{\texttt{circ}}$ where group sifting during ROBDD construction is deactivated. One can see that the cactus plots for HSSATSolve$^{\texttt{circ,ns}}$ are pretty similar in Figs. 6, 7, and 8. This can be easily explained: The variables in the formulas of types Ɐ∃, Ɐ, and ∃Ɐ occur in the same order, see Eqns. (5), (7), and (6), they only differ from their quantifier types. The matrices of the formulas are (almost) identical ($\neg f_{IM}$ for Eqn. (5), f_{IM} for Eqn. (7), $f_{IM} \wedge (\vec{K} \neq \vec{K}_{\texttt{orig}})$ for Eqn. (6)). Thus, HSSATSolve$^{\texttt{circ,ns}}$ builds similar ROBDDs with the same variable order which is not changed during the construction. Since the run times of our approach are dominated by the ROBDD construction and evaluation is very fast in comparison, the differences between the results of HSSATSolve$^{\texttt{circ}}$ in Figs. 6 and 7 can be explained by the fact that group sifting with a single group of variables in formulas of type Ɐ (Fig. 7) has more degrees of freedom than with two groups of variables for formulas of type Ɐ∃ (Fig. 6).

For our formulas of type ∃Ɐ, HSSATSolve$^{\texttt{circ}}$ outperforms the existing solvers, see Fig. 8. Among the other solvers ElimSSAT performs best. ElimSSAT is well suited for our formulas with input probabilities of 0.5. In general, it is only suited for input probabilities which are a sum of negative powers of two [35]. If we have other probabilities for the inputs $\vec{X}$ in Eqn. (6), then the input probabilities have to be rounded and ElimSSAT produces errors of unknown and uncontrollable size. HSSATSolve can work with arbitrary rational numbers as input probabilities and it works with unlimited precision.

Formulas of type H compute the fraction of keys with high criticality, see Eqn. (8). Here we first investigate the effect of group sifting and of using logic synthesis by ABC before building ROBDDs. In HSSATSolve$^{\texttt{circ,nABC}}$ ABC is omitted, in HSSATSolve$^{\texttt{circ,ns}}$ group sifting is omitted, and in HSSATSolve$^{\texttt{circ,nsnABC}}$ both techniques are omitted. For this evaluation we fixed the criticality bound c to 0.999. (Note that there is basically no run time difference for our method with different criticality bounds. Different values of c do change the value of the result, of course, but not the evaluation steps which have to be performed by Alg. 1.) Results for formulas of type H can be found in Fig. 9. The results show that the

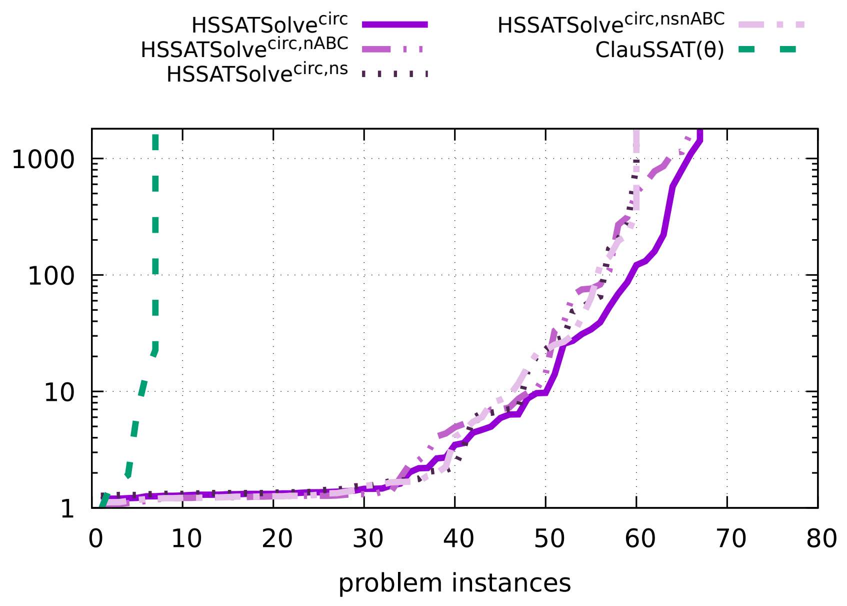


Figure 9 Type H.

effect of ABC is less significant than the effect of group sifting. Whereas ABC apparently accelerates the overall run times of the tool, it does not change the number of benchmarks which are solved before the timeout. Group sifting helps on the other hand, as already shown in the other experiments. Formulas of type H also lie in the class SSAT(Θ) recently proposed by Fan and Jiang [11]. They presented the prototype tool ClauSSat(Θ) for solving SSAT(Θ) formulas. However, Fig. 9 shows that ClauSSat(Θ) is currently not well suited for formulas from our application domain.

In Figs. 10 and 11 we demonstrate how our tool can be used for quality assessment in logic locking.

In a first experiment the criticality bound c of the keys is again fixed to 0.999 (which would mean for an application that we consider a key only as highly critical, if it produces correct outputs for at least 99,9% of the possible input combinations). We investigate how increasing the key lengths influences the average fraction of keys with high criticality. Fig. 10 shows results for benchmark circuits b01, b03, b05, b06, and b13. For the chosen logic locking method and the considered circuits the fraction of keys with high criticality rapidly decreases with increasing key lengths (chosen in steps from 4 to 32). Since there is at least one unlocking key, the fraction of highly critical keys can of course not be lower than $\frac{1}{2^{kl}}$ with key length kl. The results show e.g. that for circuit b05 the logic locking is not absolutely perfect, since for key length 16 there is not only one, but there are 4 critical keys (out of $2^{16} = 65536$ possible keys), for key length 32 there are 32 critical keys (out of 2^{32} possible keys).

Fig. 11 shows (for a fixed key length of 16) how the fraction of keys with high criticality changes, if we change our notion of "high criticality". If we choose a criticality bound $c = 80\%$, then circuit c3540, e.g., still has 1138 keys (a fraction of approximately 1.7%) with high criticality. Nevertheless, all of the considered circuits reach the lowest possible fraction of $\frac{1}{2^{16}}$ for key length 16, if we consider a key only as critical, if it unlocks the circuit completely ($c = 1.0$).

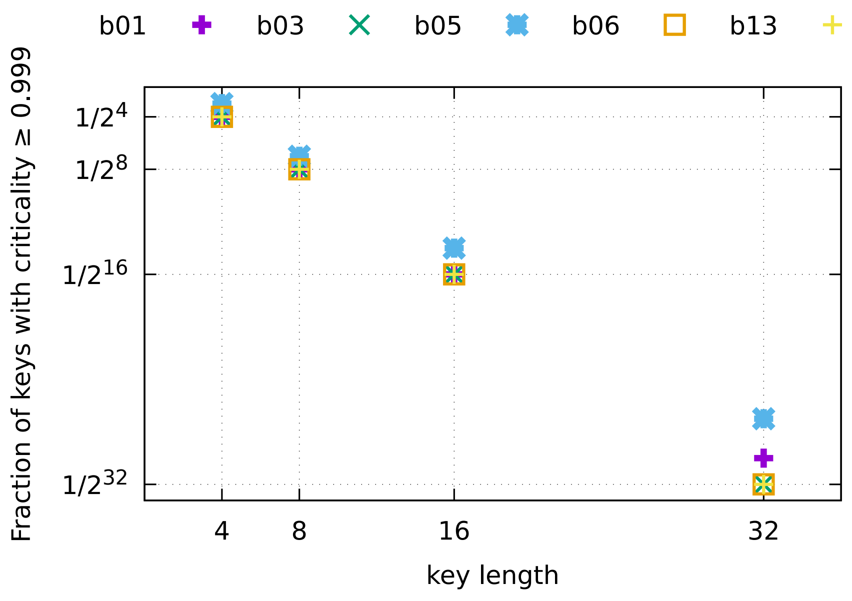


Figure 10 Fraction of keys with criticality ≥ 0.999, key lengths 4, 8, 16, 32.

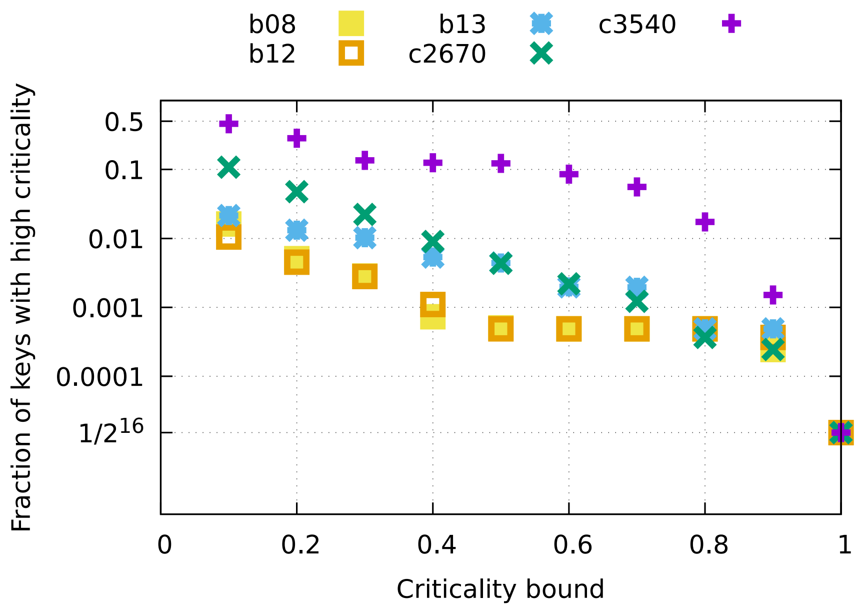


Figure 11 Fraction of keys with criticality $\geq 0.1, 0.2, \ldots, 0.8, 0.9, 1.0$, key length 16.

6 Conclusions and Future Work

We introduced the problem of Hierarchical Stochastic SAT, we presented a prototype solver for HSSAT, and we used the solver to investigate interesting questions in quality assessment for logic locking methods. For some subclasses of HSSAT formulas for which solvers already exist, the new solver compares favorably with the existing solvers. In the future, we plan to use our new tool to compare the quality of different existing logic locking schemes. We also plan to develop HSSAT algorithms going beyond ROBDD construction. Finally, we plan to add certification to the solver.

References

1 Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. Preventing IC piracy using reconfigurable logic barriers. *IEEE Des. Test Comput.*, 27(1):66–75, 2010. doi:10.1109/MDT.2010.24.

2 Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. available at https://people.eecs.berkeley.edu/~alanmi/abc/, 2019.

3 Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010. doi:10.1007/978-3-642-14295-6_5.

4 Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings of IEEE International Symposium Circuits and Systems (ISCAS 85)*, pages 677–692. IEEE Press, Piscataway, N.J., 1985.

5 Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.

6 Pei-Wei Chen, Yu-Ching Huang, and Jie-Hong R. Jiang. A sharp leap from quantified boolean formula to stochastic boolean satisfiability solving. 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 3697–3706. AAAI Press, 2021. doi:10.1609/AAAI.V35I5.16486.

7 Gustavo K. Contreras, Md. Tauhidur Rahman, and Mohammad Tehranipoor. Secure split-test for preventing IC piracy by untrusted foundry and assembly. In *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFTS 2013, New York City, NY, USA, October 2-4, 2013*, pages 196–203. IEEE Computer Society, 2013. doi:10.1109/DFT.2013.6653606.

8 Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. Rt-level itc'99 benchmarks and first ATPG results. *IEEE Des. Test Comput.*, 17(3):44–53, 2000. doi:10.1109/54.867894.

9 Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. A novel hardware logic encryption technique for thwarting illegal overproduction and hardware trojans. In *2014 IEEE 20th International On-Line Testing Symposium, IOLTS 2014, Platja d'Aro, Girona, Spain, July 7-9, 2014*, pages 49–54. IEEE, 2014. doi:10.1109/IOLTS.2014.6873671.

10 Yu-Wei Fan and Jie-Hong R. Jiang. Sharpssat: A witness-generating stochastic boolean satisfiability solver. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 3949–3958. AAAI Press, 2023. doi:10.1609/AAAI.V37I4.25509.

11 Yu-Wei Fan and Jie-Hong R. Jiang. Unifying decision and function queries in stochastic boolean satisfiability. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 7995–8003. AAAI Press, 2024. `doi:10.1609/AAAI.V38I8.28637`.

12 The gnu multiple precision arithmetic library. `https://gmplib.org/`, 2024.

13 Markus Hecher and Johannes Fichte. Model counting competition 2023. `https://mccompetition.org/`, 2023.

14 Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints An Int. J.*, 21(1):41–58, 2016. `doi:10.1007/S10601-015-9204-Z`.

15 Tuukka Korhonen and Matti Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters (short paper). In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 8:1–8:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPICS.CP.2021.8`.

16 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 751–757. IJCAI/AAAI Press, 2016. URL: `http://www.ijcai.org/Abstract/16/112`.

17 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. `doi:10.24963/IJCAI.2017/93`.

18 Stephen M. Majercik and Byron Boots. DC-SSAT: A divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 416–422. AAAI Press / The MIT Press, 2005. URL: `http://www.aaai.org/Library/AAAI/2005/aaai05-066.php`.

19 Marcel Merten, Sebastian Huhn, and Rolf Drechsler. Quality assessment of rfet-based logic locking protection mechanisms using formal methods. In *IEEE European Test Symposium, ETS 2022, Barcelona, Spain, May 23-27, 2022*, pages 1–2. IEEE, 2022. `doi:10.1109/ETS54262.2022.9810459`.

20 Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In Richard L. Rudell, editor, *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 74–77. IEEE Computer Society / ACM, 1995. `doi:10.1109/ICCAD.1995.479994`.

21 Jeyavijayan Rajendran, Youngok K. Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 83–89. ACM, 2012. `doi:10.1145/2228360.2228377`.

22 Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok K. Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Trans. Computers*, 64(2):410–424, 2015. `doi:10.1109/TC.2013.193`.

23 Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. EPIC: ending piracy of integrated circuits. In Donatella Sciuto, editor, *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 1069–1074. ACM, 2008. `doi:10.1109/DATE.2008.4484823`.

24 Ricardo Salmon and Pascal Poupart. On the relationship between satisfiability and markov decision processes. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 1105–1115. AUAI Press, 2019. URL: `http://proceedings.mlr.press/v115/salmon20a.html`.

25 Christoph Scholl, Tobias Seufert, and Fabian Siegwolf. Hierarchical SSAT: Benchmarks and solver HSSATSolve. `https://nc.informatik.uni-freiburg.de/index.php/s/PQ9zBAejC2ERCTD`, 2024.

26 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1169–1176. ijcai.org, 2019. `doi:10.24963/IJCAI.2019/163`.

27 Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020. `doi:10.1007/978-3-030-53288-8_24`.

28 Fabio Somenzi. Efficient manipulation of decision diagrams. *Int. J. Softw. Tools Technol. Transf.*, 3(2):171–181, 2001. `doi:10.1007/S100090100042`.

29 Mate Soos and Kuldeep S. Meel. Arjun: An efficient independent support computation technique and its applications to counting and sampling. In Tulika Mitra, Evangeline F. Y. Young, and Jinjun Xiong, editors, *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022*, pages 71:1–71:9. ACM, 2022. `doi:10.1145/3508352.3549406`.

30 Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976. `doi:10.1016/0304-3975(76)90061-X`.

31 Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 137–143. IEEE Computer Society, 2015. `doi:10.1109/HST.2015.7140252`.

32 Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using SAT solving in components. Technical Report SIG-FPAI-103-B506, JSAI Technical Report, March 2017. in Japanese.

33 G. S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer Berlin Heidelberg, 1983.

34 Klaus W. Wagner. The complexity of combinatorial problems with succinct input representation. *Acta Informatica*, 23(3):325–356, 1986. `doi:10.1007/BF00289117`.

35 Hao-Ren Wang, Kuan-Hua Tu, Jie-Hong Roland Jiang, and Christoph Scholl. Quantifier elimination in stochastic boolean satisfiability. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 23:1–23:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPICS.SAT.2022.23`.

36 Yang Xie and Ankur Srivastava. Mitigating SAT attack on logic locking. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2016. `doi:10.1007/978-3-662-53140-2_7`.

37 Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. Sarlock: SAT attack resistant logic locking. In William H. Robinson, Swarup Bhunia, and Ryan Kastner, editors, *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, pages 236–241. IEEE Computer Society, 2016. `doi:10.1109/HST.2016.7495588`.

38 Muhammad Yasin, Jeyavijayan (JV) Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(9):1411–1424, 2016. doi:10.1109/TCAD.2015.2511144.

39 Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1601–1618. ACM, 2017. doi:10.1145/3133956.3133985.

40 Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrión Schäfer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan Rajendran. What to lock?: Functional and parametric locking. In Laleh Behjat, Jie Han, Miroslav N. Velev, and Deming Chen, editors, *Proceedings of the on Great Lakes Symposium on VLSI 2017, Banff, AB, Canada, May 10-12, 2017*, pages 351–356. ACM, 2017. doi:10.1145/3060403.3060492.

A Proof of Lemma 12

Proof. Let Φ be a closed HSSAT formula with matrix $\texttt{matrix}(\Phi) = \phi$ which is recursively evaluated according to Def. 5 and let x be an existential variable in its prefix.

Let α be an assignment over $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$. Here we use the notation $\Phi|_\alpha$ for the closed HSSAT formula which replaces $\texttt{matrix}(\Phi)$ by its cofactor $\texttt{matrix}(\Phi)|_\alpha$, removes all comparisons according to case (d) of Def. 3 at hierarchy levels $> \texttt{hlevel}(x)$, and removes all variables $y_i \in \{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ together with their quantifiers $Q_i \in \{\exists, \forall, \text{Ɐ}^{p_i}\}$ from the prefix of Φ.

$\Phi|_\alpha = (\exists x \Phi')$ is a subproblem occurring during the recursive evaluation of Φ according to Def. 5 with $\texttt{matrix}(\Phi|_\alpha) = \phi|_\alpha$.

Let x be defined in terms of $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ in ϕ and for all hl with $1 \leq hl \leq \texttt{hlevel}(x)$ the equation "$0\ \texttt{op}(hl)\ \texttt{prob}(hl)$" does not hold.

We make a case distinction wrt. the cofactors $(\phi|_\alpha)|_x$ and $(\phi|_\alpha)|_{\neg x}$ of $\phi|_\alpha$:

- Case 1: Both $(\phi|_\alpha)|_{\neg x}$ and $(\phi|_\alpha)|_x$ do not represent the constant 0 function.
  This means that there are two full assignments $\beta : \texttt{BV}(\Phi) \to \{0,1\}$ and $\beta' : \texttt{BV}(\Phi) \to \{0,1\}$ with $\beta(y) = \beta'(y) = \alpha(y)$ for all $y \in \texttt{BV}(\Phi)$ with $\pi^{-1}(y) < \pi^{-1}(x)$, $\beta(x) = 1$, $\beta'(x) = 0$, and $\beta[\phi] = \beta'[\phi] = 1$. This is a contradiction to the assumption that x is defined in terms of $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ in ϕ, since β and β' both satisfy ϕ and agree on $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$, but $\beta(x) \neq \beta'(x)$. Thus, the case assumption cannot hold, if x is defined in terms of $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ in ϕ.
- Case 2: $(\phi|_\alpha)|_{\neg x}$ represents the constant 0 function, $(\phi|_\alpha)|_x$ not.
  Since $\texttt{matrix}(\Phi'|_{\neg x}) = (\phi|_\alpha)|_{\neg x}$ represents the constant 0 function and for all hl with $1 \leq hl \leq \texttt{hlevel}(x)$ the equation $0\ \texttt{op}(hl)\ \texttt{prob}(hl)$ does not hold, we obtain $\textsc{Eval}(\Phi'|_{\neg x}) = 0$. We show $\textsc{Eval}(\Phi'|_{\neg x}) = 0$ by proving $\textsc{Eval}(\Psi) = 0$ for all subproblems Ψ of $\Phi'|_{\neg x}$ occurring during the recursive evaluation of $\Phi'|_{\neg x}$ according to Def. 5. We show this by induction over the sum $s := \texttt{hlevel}(\Psi) + |\texttt{BV}(\Psi)|$. If $s = 0$, then either case (1) or case (2) of Def. 5 applies to Ψ. Since $f_{(\phi|_\alpha)|_{\neg x}} = 0$ by case assumption, case (2) of Def. 5 cannot occur. We are in case (1) and $\textsc{Eval}(\Psi) = 0$.
  For cases (3), (4), and (5) of Def. 5 we have $\Psi = \exists z \Psi'$, $\Psi = \forall z \Psi'$, or $\Psi = \text{Ɐ}^p z \Psi'$. We can assume $\textsc{Eval}(\Psi|_{\neg z}) = \textsc{Eval}(\Psi|_z) = 0$ by induction hypothesis and thus $\textsc{Eval}(\Psi) = 0$.
  For case (6) we consider $\Psi = (\Psi'\ \texttt{op}(\texttt{hlevel}(\Psi))\ \texttt{prob}(\texttt{hlevel}(\Psi)))$. We can assume by induction hypothesis that $\textsc{Eval}(\Psi') = 0$. Now we need our additional restriction on the choice of defined existential variables to ensure that the comparison according to Def. 5,

case (6), is not able to change the value 0 of $\textsc{Eval}(\Psi')$ into a different value. We have $1 \leq \texttt{hlevel}(\Psi) \leq \texttt{hlevel}(x)$, therefore $0\ \texttt{op}(\texttt{hlevel}(\Psi))\ \texttt{prob}(\texttt{hlevel}(\Psi))$ does not hold and thus $\textsc{Eval}(\Psi) = 0$.
Now we have $\textsc{Eval}(\Phi'|_{\neg x}) = 0$. According to Def. 5

$$\textsc{Eval}(\exists x \Phi') = \max(\textsc{Eval}(\Phi'|_{\neg x}), \textsc{Eval}(\Phi'|_x)) = \max(0, \textsc{Eval}(\Phi'|_x)) = \textsc{Eval}(\Phi'|_x).$$

Altogether we have

$$\textsc{Eval}(\Phi|_\alpha) = \textsc{Eval}(\exists x \Phi') = \textsc{Eval}(\Phi'|_x). \tag{10}$$

Consider a definition ψ of x by $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ in ϕ and a full assignment $\beta : \texttt{BV}(\Phi) \to \{0,1\}$ with $\beta(y) = \alpha(y)$ for all $y \in \texttt{BV}(\Phi)$ with $\pi^{-1}(y) < \pi^{-1}(x)$, $\beta(x) = 1$ and $\beta[\phi] = 1$. Such an assignment β exists, since $(\phi|_\alpha)|_x$ does not represent the constant 0 function. Since ψ is a definition, we have $\psi[\beta] = \beta(x) = 1$. Since ψ only depends on variables from $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$, we also have $\psi[\alpha] = \beta(x) = 1$. Thus, we have

$$\textsc{Eval}(\Phi[x := \psi]|_\alpha) = \textsc{Eval}(\Phi'[x := \psi[\alpha]]) = \textsc{Eval}(\Phi'|_x). \tag{11}$$

From Eqns. (10) and (11) we conclude

$$\textsc{Eval}(\Phi|_\alpha) = \textsc{Eval}(\Phi[x := \psi]|_\alpha). \tag{12}$$

- Case 3: $(\phi|_\alpha)|_x$ represents the constant 0 function, $(\phi|_\alpha)|_{\neg x}$ not.
  We conclude $\textsc{Eval}(\Phi|_\alpha) = \textsc{Eval}(\Phi[x := \psi]|_\alpha)$ similar to Case 2.
- Case 4: Both $(\phi|_\alpha)|_{\neg x}$ and $(\phi|_\alpha)|_x$ represent the constant 0 function.
  Similar to Case 2 we obtain $\textsc{Eval}(\Phi'|_{\neg x}) = 0$ and $\textsc{Eval}(\Phi'|_x) = 0$. This leads to

$$\textsc{Eval}(\Phi|_\alpha) = \textsc{Eval}(\exists x \Phi') = 0. \tag{13}$$

  We have

$$\textsc{Eval}(\Phi[x := \psi]|_\alpha) = \textsc{Eval}(\Phi'[x := \psi[\alpha]]) = 0 \tag{14}$$

  for arbitrary formulas ψ over $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$ and thus also for each definition of x.

Case 1 cannot occur and in the remaining cases we have $\textsc{Eval}(\Phi|_\alpha) = \textsc{Eval}(\Phi[x := \psi]|_\alpha)$ for arbitrary assignments α over $\{y \in \texttt{BV}(\Phi) \mid \pi^{-1}(y) < \pi^{-1}(x)\}$. This proves $\textsc{Eval}(\Phi) = \textsc{Eval}(\Phi[x := \psi])$. ◀

Trusted Scalable SAT Solving with On-The-Fly LRAT Checking

Dominik Schreiber
Karlsruhe Institute of Technology, Germany

Abstract

Recent advances have enabled powerful distributed SAT solvers to emit proofs of unsatisfiability, which renders them as trustworthy as sequential solvers. However, this mode of operation is still lacking behind conventional distributed solving in terms of scalability. We argue that the core limiting factor of such approaches is the requirement of a single, persistent artifact at the end of solving that is then checked independently (and sequentially). As an alternative, we propose a bottleneck-free setup that exploits recent advancements in producing and processing LRAT information to immediately check all solvers' reasoning *on-the-fly* during solving. In terms of clause sharing, our approach transfers the guarantee of a derived clause's soundness from the sending to the receiving side via cryptographic signatures. Experiments with up to 2432 cores (32 nodes) indicate that our approach reduces the running time overhead incurred by proof checking by an order of magnitude, down to a median overhead of $\leq 42\%$ over non trusted solving.

2012 ACM Subject Classification Hardware → Theorem proving and SAT solving; Theory of computation → Automated reasoning; Computing methodologies → Distributed algorithms

Keywords and phrases SAT solving, distributed algorithms, proofs

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.25

Supplementary Material *Collection (Experimental data and links to software)*: `https://github.com/domschrei/mallob-impcheck-data`
archived at `swh:1:dir:d1ea41171e7e9f185b5930797c61a5be40dc0f6b`

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Acknowledgements The author thanks Marijn J. H. Heule, who inspired and encouraged this work; Florian Pollitt, Mathias Fleury, and Armin Biere who enabled this work with their enhancements to CaDiCaL [39]; Peter Sanders for helpful discussions and valuable remarks; Patrick Hegemann for double-checking cryptographic details; and all anonymous reviewers for their valuable feedback.

1 Introduction

The Boolean Satisfiability (SAT) problem, i.e., to satisfy a given Boolean expression or to report its unsatisfiability, is an essential building block at the core of automated reasoning, symbolic AI, and formal verification [18]. Due to its high practical relevance, increasingly efficient SAT solving approaches have emerged over the last decades, prompting a plethora of applications [11, 34, 35, 41, 42, 49, 52] to use SAT solvers as efficient blackbox engines. Today, researchers and industrial users increasingly aim to exploit distributed environments [9, 13, 20] to push the frontier of problems that are feasible to solve.

An important topic in SAT research is the reliability and trustworthiness of solvers [6, 8, 33, 36]. In particular, sequential solvers have been able for many years to output *proofs of unsatisfiability* – witnesses for a formula's unsatisfiability that can be verified by independent and even formally verified *proof checkers* [30]. For the longest time, the best performing parallel and distributed solvers have been missing this crucial feature (*cf.* [26]). Only recently, Michaelson et al. proposed a feasible approach to produce proofs with parallel and distributed clause-sharing SAT solving [36]. The technology enabling this approach is a clausal proof format called LRAT [14], where each clause derivation has a unique ID and explicitly references the prior clauses required to check the derivation. This information allows to feasibly reconstruct a single proof from many LRAT-producing SAT solvers that run in parallel and periodically exchange clauses in an *all-to-all* fashion [36].

Even after these advances, the scalability of trustworthy parallel and distributed solving remains lacking. In Michaelson et al.'s procedure, all relevant proof information is funneled into a single process and then output *sequentially.* As such, the production and checking of a combined proof is invariably throttled by the (I/O) bandwidth at the final process. Likewise, to our knowledge all LRAT checkers so far are sequential. Note that proof size increases both with solving time and, to a lesser degree, the number of solvers [36]. Maintaining the active clauses in huge proofs may further slow down the proof checker or even cause main memory shortage at some point. For all these reasons put together, we argue that the production and checking of proofs currently poses a distinct bottleneck for trusted distributed solving.

To address this conceptual problem, we believe that it is sensible to slightly relax the task at hand. Even if trusting a solver's result is crucial, a witness for this result in the shape of a persistent artifact may often be expendable – for instance, in continuous software verification [10]. Dropping this requirement, we are able to replace Michaelson et al.'s three-stage procedure with a *single-stage* procedure that finishes whenever solving finishes (see Fig. 1). We build upon a suggestion by Marijn Heule to check proofs *during solving* via inter-process communication and raise this idea to a scalable level: Each solver thread now forwards each derivation to a checker process, which checks the derivation at once. To transfer the guarantee of a clause's soundness from one checker to another, each checker outputs a cryptographic *signature* for each clause it checked. The receiving solver's checker then validates the signature by recomputing it. Given a secret key shared among all checkers, we show this procedure to be as trustworthy as *post mortem* proof checking: Any bug or error that happens to create an *unsound* clause with a *valid* signature is substantially less likely than a trusted checker program is to "hallucinate" unsatisfiability due to a hardware-side memory error. Our critical code is a small, isolated, and simple C99 codebase whose formal verification we believe to be a distinct possibility in the near future.

We implemented our approach using the LRAT-producing solver CaDiCaL [39] and the distributed system MallobSat [44, 45]. For a fair comparison, we applied a number of updates to MallobSat with proof production [36]. Experiments on up to 2432 cores (32 nodes) indicate that our approach incurs a median slowdown of 40% (at 76 cores) to 42% (at 2432 cores) over conventional, non trusted solving. In comparison, the median slowdown of

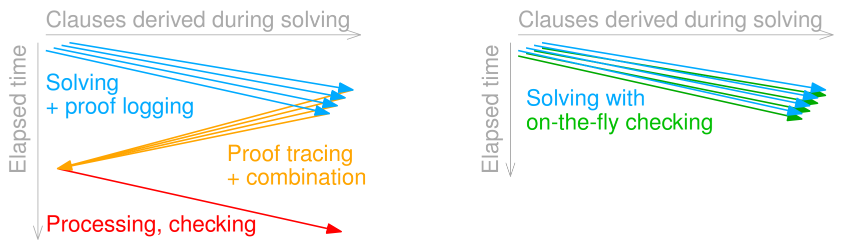


Figure 1 Schematic overview on a prior approach [36] (left) and our new approach (right) to trusted parallel solving. Each line, very roughly speaking, corresponds to an execution thread.

producing and checking a combined proof surpasses 330% at 76 cores and 750% at 1216 cores. As such, our approach's overhead is smaller by an order of magnitude and can enable trusted solving in cases where explicit proof production is infeasible. We anticipate a number of use cases for this technology, ranging from debugging parallel solvers over distributed solving with error-prone or insecure communication to formally verified scalable SAT solving.

In Section 2, we introduce preliminaries and prior work on the subject. We describe our approach in Section 3 and present an experimental evaluation in Section 4. In Section 5, we discuss the possible ramifications of our contributions. Section 6 concludes our work.

2 Preliminaries

We discuss some relevant preliminaries to the work at hand with a focus on proofs of unsatisfiability in sequential, parallel, and distributed SAT solving.

2.1 SAT Solving

We consider formulas in *conjunctive normal form* (CNF), i.e., of the form $F = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{n_i} l_{ij}$, where each of the k disjunctions is called a *clause of length* n_i and each *literal* l_{ij} is a Boolean variable or its negation. The *Boolean satisfiability* (SAT) problem is to assign a value to all Boolean variables in F such that F evaluates to `true` or to recognize that this is impossible. F is called *satisfiable* if such an assignment exists and *unsatisfiable* otherwise. We say that a clause c is *sound* (w.r.t. F) if and only if c is a logical consequence of F, i.e., $F \wedge \neg c$ is unsatisfiable. As a special case, F is unsatisfiable if and only if clause $c = \emptyset$ is sound w.r.t. F.

Today's most popular and most efficient sequential SAT solvers build upon the *Conflict-Driven Clause Learning* (CDCL) paradigm: The solver searches the space of partial variable assignments and derives a *redundant conflict clause* whenever it encounters a conflict with the current assignment [15]. These redundant clauses are crucial for pruning search space and for eventually deriving the *empty clause*, i.e., the contradiction in an unsatisfiable formula.

The currently most scalable parallel and distributed SAT solvers are *clause-sharing portfolio solvers*, where many (mostly CDCL) solver threads run in parallel on the original formula and, crucially, periodically exchange promising learned conflict clauses [2]. We recently confirmed that clause sharing is in fact the main driver of the scalability of the state-of-the-art distributed solver MallobSat and can achieve decent performance even if running *identical*, i.e., non diversified solver threads [45]. There are other parallelization approaches to SAT with explicit search space splitting [25, 48], which we do not cover here.

We refer to the author's dissertation [43] for a more detailed introduction to sequential, parallel, and distributed SAT solving in the context of the work at hand.

2.2 Proofs of Unsatisfiability

If a formula F is satisfiable, any found satisfying assignment can be verified in linear time by evaluating F under the assigned values. Since today's solvers commonly output such a satisfying assignment when reporting satisfiability, we do not consider their trustworthiness to be a notable issue for satisfiable problems. By contrast, if F is unsatisfiable and a justification for the claimed result is desired, the solver must produce a *proof of unsatisfiability*. Such a proof contains the solver's chain of reasoning that lead to the empty clause. Proofs can be checked by independent and sometimes formally verified *proof checkers* [30]. They can also be useful for reasons beyond trust, e.g., for Minimally Unsatisfiable Subset (MUS) extraction [3] or for analyzing the efficiency [47] or scalability [28] of solving approaches.

In this work we focus on the LRAT (*Linear Reverse Asymmetric Tautology*) clausal proof format [14]. Simply put, LRAT proof information output by a solver reflects the solver's changes to its clause set: If a new clause is derived, a *clause addition* (or *derivation*) is appended to the proof, and if a clause is discarded, a corresponding *deletion statement* is appended to the proof. The LRAT format requires each derivation of a clause c to be labeled with a unique ID, $id(c)$, and to explicitly include the required *dependencies* for this derivation, often called *hints*, in the form of a sequence D_c of clause IDs. A deletion statement features a sequence of IDs that refer to the clauses to delete.

LRAT proof checkers traverse the proof information at hand in a single linear pass. A checker first receives all original problem clauses F and initializes its own clause set $C := F$. It then successively applies the proof's individual clause additions and deletions to C. Clause deletions are strictly speaking not required but are important in practice since they significantly reduce the memory footprint of proof checking [23]. Each clause addition is checked using the *LRAT criterion* or, in many cases, a simpler special case named the *LRUP criterion*. While the former corresponds to the most powerful proof format known [29], the latter currently covers most solvers' reasoning, including CaDiCaL's (which we use for our study) [39]. For a derivation of clause c with dependencies D_c, LRUP requires that the clause set $\{\{\bar{l}\} : l \in c\} \cup \{c' : id(c') \in D_c,\ c' \in C\}$ results in an efficiently computable conflict: Asserting the negated literals of c, the sequence of clauses referenced by D_c must break down into unit clauses and, finally, the empty clause. As such, the checker confirms that $C \wedge \neg c$ results in unsatisfiability. Therefore, c is sound w.r.t. C. The derivation of the empty clause $\tilde{c} := \emptyset$ poses a special case where the unsatisfiability of C, and hence F, is testified.

Few parallel solvers support proof production. One of them is Gimsatul [19], an integrated shared-memory solver written from scratch. Its architecture allows for outputting a single consistent DRAT proof from all solver threads. Unlike LRAT, the earlier DRAT format [22] does not feature any explicit dependency information, which renders DRAT proofs easier to produce but substantially more expensive to check [14]. Gimsatul's approach is limited to shared memory, and combining multiple DRAT proofs into a single proof has, so far, largely resulted in proofs that are infeasible to check [26, 36].

Michaelson et al. [36] recently proposed a more general approach to proof production that is viable in distributed environments. It features the following stages:

1. **Solving**: All solver threads write their LRAT proof information to individual *partial proof* files. Clause IDs are assigned in a globally unique manner: All clause IDs produced by a certain solver are pairwise congruent modulo the total number of solvers.
2. **Combination**: As soon as a solver finds the empty clause, the solving procedure is *rewound*. All solver threads read their respective partial proofs in reverse and trace all transitive dependencies of the found empty clause. Each clause sharing operation is reversed by redistributing the IDs of required *remote* clauses back to their origin. All clause derivations identified as required are funneled into a single process, which writes the combined and dependency-ordered LRAT proof information to a single file. Note that this information cannot be streamed to a checker directly since it is still reversed.
3. **Checking**: The combined proof is read *from back to front*, which amounts to the correct chronological order, and is validated by a sequential LRAT checker.

In its original form, this proof production approach featured additional pre– and postprocessing stages (such as exhaustive unit propagation prior to solving), which were required due to limitations of the used solver backend [36]. Since then, Pollitt et al. published a version of CaDiCaL with full LRAT support [39] that renders these steps obsolete. We outline according updates of Michealson et al.'s approach in some more detail in Section 4.2.1.

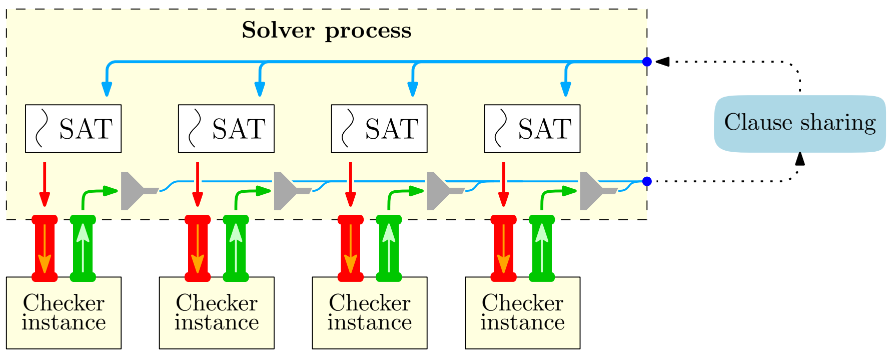


Figure 2 Information flow in a solver process with our approach. Each SAT solver thread emits LRAT proof information, which is streamed to the solver thread's corresponding checker instance (red pipe downwards). A checker produces *signatures* for checked clauses (green pipe upwards). Some of the *signed clauses* are exported and exchanged across processes. Incoming shared clauses are forwarded to the corresponding solvers. Successfully imported clauses, together with their signatures, are as well streamed to the respective checker.

3 Trusted Solving Approach

We now present our approach to trusted parallel and distributed solving.

3.1 Overview

Our approach exploits a suggestion by Marijn Heule for the setting of sequential SAT solving: Given a SAT solver that "natively" produces LRAT information [39], its proof output can be redirected to a concurrently running LRAT checking process that checks the solver's reasoning *on-the-fly*.[1] This can be achieved with inter-process communication (e.g., UNIX pipes) and hence without writing proof information to disk. Note that this method is not necessarily viable for the earlier and still more widely used DRAT proof format [22] since DRAT checking is substantially more expensive and therefore likely to stall solving.

Our distributed solver employs p *solver threads* distributed across m *processes*, with $t = p/m$ solver threads per process. Each solver thread produces a stream of LRAT proof information, assigning globally unique clause IDs just like in Michaelson et al.'s approach (Section 2.2). We suggest to perform the on-the-fly checking outlined above for each solver thread, running one associated *checker instance* (i.e., LRAT checking process) each.

An important observation for our approach is that a sound clause is sound *globally*: Once established to be implied by the input formula, a clause can safely be used regardless of its particular dependencies or a checker's internal state. As such, once a derived clause has been checked by some checker instance, other checker instances can add the clause without checking, i.e., as if it were an *axiom* or an original problem clause. The clause can then be referenced in subsequent derivations just like locally produced clauses.

Fig. 2 illustrates the resulting solving and checking setup. All produced clause derivations are first redirected to the solver's associated checker instance. Clauses can only be exported and shared after they have been checked. (This is ensured not only by the proposed architecture but also by a *signing* mechanism that we explain in detail further below.)

[1] `https://github.com/marijnheule/coch-demo`

By checking each clause *before* exporting it to other solvers, it appears obvious that each clause used in the distributed solving procedure has been checked at some point, and one may argue that this is sufficient to render distributed solving trustworthy. However, we do not want to rely on the correctness of distributed clause sharing and its underlying buffering and communication mechanisms. For instance, a single flipped bit in a clause sharing buffer can result in an allegedly sound clause that in fact induces a wrong result. Perhaps a more realistic scenario, a programming error that (under rare circumstances) causes reading of a buffer beyond its limit can result in "garbage clauses". As an example, the distributed solver MallobSat [45] features multithreading, multi-processing, inter-process communication with UNIX pipes and shared memory, and distributed asynchronous message passing. All of these mechanisms are involved in clause sharing. Programming such systems can be challenging, and formally verifying distributed programs is laborious and only covers selected aspects of the used technology stack [32, 51]. Even if application code is provably correct, implementations of the Message Passing Interface (MPI) are known to suffer from bugs [16]. For these reasons put together, we suggest to not make *any* assumptions on the behavior nor the correctness of a distributed solver and its underlying communication mechanisms.

Our aim is to transfer the guarantee of a clause's soundness from one checker instance to another without relying on correct transmission of clauses. To this end, we have each checker module emit a *signature* $\mathcal{S}(c)$ for a checked produced clause c. Just like its LRAT clause ID [36], this signature is considered part of the clause during sharing. At the receiving side, the checker module validates the incoming clause's signature upon its import. Under the guarantee that only trusted modules are able to compute clause signatures, we show that this method grants full confidence in the soundness of all clauses leaving a solver-checker unit. We also use the same signing mechanism for a second purpose, namely to ensure that all solvers indeed operate on the same, intended input formula.

3.2 Interface

Our solving procedure features three kinds of trusted modules: one *parser* at one particular process; p *checkers* $I_1, \dots, I_p$ (t per process, i.e., one per solver thread); and, optionally, a *confirmer* at some process. We assume that our distributed solving procedure has a *private input* to these $p+2$ modules, namely a 128-bit key K that will be used to compute signatures.[2] K is handed to our trusted modules directly and must be inaccessible otherwise.

First, a **parser** takes a path to the input formula F. It parses F, computes the signature $\mathcal{S}(F)$ for F, and returns the pair $(F, \mathcal{S}(F))$, which can then be distributed to all processes.

Next, we describe the interface of each **checker** I, shown in Fig. 3. Note that this interface generalizes the LRAT proof format: `load` corresponds to specifying the formula, `produce` corresponds to adding clause derivations, and `delete` corresponds to deleting clauses.

The first call a checker expects is `init`, where the caller commits to the formula's signature $\mathcal{S}(F)$ in advance. I then receives F via `load`. Our interface supports multiple subsequent calls to `load`, which allows reading large formulas in chunks. Loading F must be concluded with `end_load`, at which point I recomputes the signature on the loaded literals to support that the loaded formula is indeed the one parsed before. The remaining methods are used during solving to process LRAT proof information. At each call to `produce`, the flag `share` indicates whether the produced clause c is intended to be shared. If this flag

[2] We do not consider the problem of secure key distribution (e.g., [53]). This problem heavily depends on choosing a specific attacker model, which is not the focus of our work.

```
Requires UNINITIALIZED - Ensures INITIALIZING
init(sig: Signature) → void
Requires INITIALIZING - Ensures INITIALIZING
load(formula: ClauseSet) → void
Requires INITIALIZING - Ensures VALID or INVALID according to output
end_load() → bool
Requires VALID - Ensures VALID or INVALID according to output
produce(id: ID, lits: Clause, hints: IDList, share: bool) → (bool, Signature?)
Requires VALID - Ensures VALID or INVALID according to output
import(id: ID, lits: Clause, sig: Signature) → bool
Requires VALID - Ensures VALID or INVALID according to output
delete(ids: IDList) → bool
Requires VALID - Ensures VALID or INVALID according to output
validate_unsat() → (bool, Signature?)
Terminates checker module from any state
terminate() → void
```

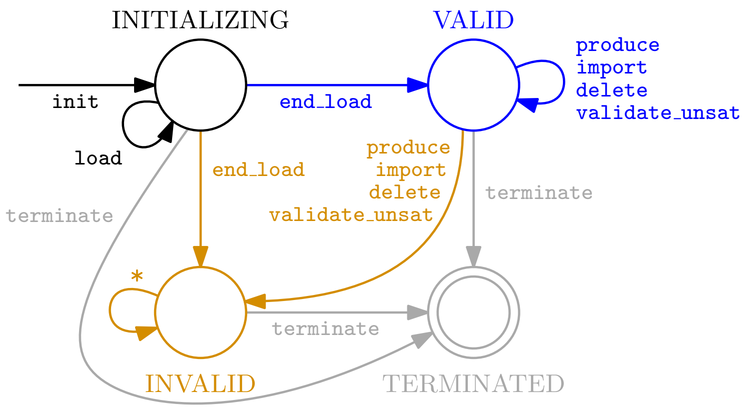


Figure 3 Checker interface with basic contracts (top) and corresponding state machine (bottom). Some directives can cause a VALID or INVALID state based on the operation's success. Transitions that violate a contract, e.g., calling `load` from VALID, all lead to INVALID and are omitted.

is `true`, the call returns signature $\mathcal{S}(c)$ on success. Each *incoming* clause c, before being referenced in subsequent derivations, must be introduced via `import` together with $\mathcal{S}(c)$, which the receiving checker validates by recomputing $\mathcal{S}(c)$. Deletion of clauses (`delete`) works exactly as in usual LRAT proofs. Finally, if a solver thread returns from solving and reports unsatisfiability, `validate_unsat` can be queried to confirm that the empty clause has indeed been found and validated through `load`/`endload`, `produce`, or `import`. On success, a special signature $\hat{s}$ based on $\mathcal{S}(F)$ is output, certifying the unsatisfiability of F.

All methods that allow for a failure state return a truth value indicating whether the respective operation was successful. A checker instance I that returned `false` at any point from any call will never return `true` from subsequent calls, in particular to `validate_unsat`.

Lastly, the **confirmer** can be invoked after solving to validate an alleged unsatisfiability signature $\hat{s}$. It takes $\hat{s}$ and the path to F, parses F and computes $\mathcal{S}(F)$, internally re-computes $\hat{s}$ based on $\mathcal{S}(F)$, and outputs whether it found $\hat{s}$ to be sound. We consider this step of confirmation a necessity only if the output obtained from a parallel solver cannot be trusted.

3.3 Signatures

We now describe how we define the signature function $\mathcal{S}$ in such a way that only our trusted modules can compute it – effectively ensuring that only checked clauses can be imported.

Our signature function $\mathcal{S}(\cdot) := \mathcal{S}_K(\cdot)$ is parametrized with secret key K and evaluated in three different contexts: on a formula F, on a single clause c, or to certify unsatisfiability ("$\perp$"). We define $\mathcal{S}_K$ based on cryptographic *Message Authentification Codes* (MACs) [21], where a sender and a receiver uphold a message's authenticity by computing a signature based on a shared secret key. Specifically, we use SipHash [1], a fast and popular *keyed pseudo-random function* (PRF) that only uses addition, rotation, and XOR (*"ARX"*). For highest confidence, we use the SipHash variant that produces 128-bit signatures.

Let $H_K : \{0,1\}^* \to \{0,1\}^{128}$ be the SipHash-2-4-128 function with key K and let "$||$" denote concatenation of data. We define $\mathcal{S}_K$ for the three above contexts as follows:

$$\mathcal{S}_K(F) := H_K(\,F\,), \quad \mathcal{S}_K(c) := H_K(\,id(c)\,||\,c\,||\,\mathcal{S}_K(F)), \quad \mathcal{S}_K(\perp) := H_K(\,20\,||\,\mathcal{S}_K(F)).$$

Defining $\mathcal{S}_K(c)$ relative to $\mathcal{S}_K(F)$ is optional hardening to ensure that interacting checkers indeed operate on the same key *and* the same formula. We define $\mathcal{S}_K(\perp)$ rather arbitrarily to sign the customary return code "20" for unsatisfiability. Adding $S_K(F)$ to an input does not weaken the MAC – since $H_K(x)$ is a PRF, $H_K(x\,||\,\alpha)$ is also a PRF for any α that does not leak information on K.[3] We also ensure that all inputs are unambiguous and that our definitions have disjoint inputs: F is encoded as 4-byte integers ending with a 2-byte zero, $20\,||\,\mathcal{S}_K(F)$ is $1 + 16 = 17$ bytes long, and clause signatures are based on $8 + 4|c| + 16$ bytes.

3.4 Correctness

We now establish the correctness of our approach given idealized signature guarantees.

▶ **Theorem 1.** *Consider our trusted solving setup for formula F. Assume that each string s that constitutes a valid signature for object x with key k indeed originates from computing $s \leftarrow \mathcal{S}_k(x)$ within our trusted code. If the solving setup outputs $\mathcal{S}_K(\perp)$, then F is unsatisfiable.*

Proof. We assume the above prerequisites. First, since only checkers can output $\mathcal{S}_K(\perp)$, there is a checker $\hat{I}$ that answered to a query `validate_unsat` with $\mathcal{S}_K(\perp)$. This implies that $\hat{I}$ received the empty clause $\hat{c} := \emptyset$ (a) via `load`/`end_load`, (b) via `produce_clause`, or (c) via `import_clause`.[4] Case (a) directly implies the unsatisfiability of F: `end_load` must have been called after all `load` calls but before `validate_unsat`, and `end_load` checks via $\mathcal{S}(F)$ that all loaded clauses, and $\hat{c}$ in particular, belong to F. In case (b), $\hat{I}$ checked $\hat{c}$, and in case (c), another checker I' signed $\hat{c}$ and therefore checked $\hat{c}$. In both cases (b) and (c), $\hat{c}$ was checked, which renders $\hat{c}$ sound if the clauses $D_{\hat{c}}$ in the derivation of $\hat{c}$ are sound. For each clause in $D_{\hat{c}}$ that is *not* an original problem clause, we recurse on the argument we just made for $\hat{c}$ itself. Since each derived clause c relies on $|D_c| > 0$ prior clauses and is added to a checker only *after* its checking, the dependencies across clauses must form a DAG whose source nodes are clauses added without checking nor dependencies, i.e., via `load`. As such, the derivation of $\hat{c}$ is sound if all clauses added via `load` by any contributing checker are original clauses of F. Since any contribution by a checker requires a prior call to `end_load` (see case (a)), which validates the loaded formula, all contributing checkers indeed operate on the intended F. The derivation of $\hat{c}$ is thus sound and F is in fact unsatisfiable. ◀

[3] By contradiction: Any strategy distinguishing $g(x) := H_K(x\,||\,\alpha)$ from true randomness can be modified into a distinguisher for $f(x) := H_K(x)$ just by replacing each call $g(x)$ with a call $f(x\,||\,\alpha)$.

[4] The empty clause is usually not shared since its derivation renders any further sharing obsolete.

For the reverse direction, our approach can *preserve* a sequential solver's completeness:

▶ **Theorem 2.** *Consider an error-free execution of the described trusted solving setup for formula F where one solver thread is complete. If F is unsatisfiable, then $\mathcal{S}_K(\bot)$ is output.*

Proof. Consider the checker instance $\hat{I}$ of a complete sequential solver S. $\hat{I}$ functionally subsumes a plain LRAT checker that mirrors the reasoning of S. Assuming sound proof logging, $\hat{I}$ will thus receive and validate the empty clause $\hat{c}$, which S derives eventually. S then terminates with unsatisfiability, which our setup confirms by querying `validate_unsat` in $\hat{I}$. The added capabilities of our checker over plain LRAT checking cannot obstruct any of these operations. In particular, error-free clause imports can only *accelerate* the progress towards unsatisfiability. As such, $\hat{I}$ will indeed output $\mathcal{S}_K(\bot)$. ◀

Note that any call to `import` with an *incorrect* signature functionally disables the checker, which makes it impossible to export further clauses or report unsatisfiability from the respective solver. Our underlying design decision is that we aim to notice every error and hence interrupt solving globally even if the solving procedure is in principle recoverable. In other settings, perhaps when faced with error-prone communication or exascale computing, it may be sensible to let checkers continue normally after rejecting a particular import.

3.5 Confidence

Let us now discuss the conditions causing the above assumptions to break in such a way that our trusted code reports unsatisfiability for a certain satisfiable formula F. We still assume that our $p+2$ trusted processes are sound, uncompromised, and the only actors knowing K. Under these assumptions, we can discern the following *attack vectors*:

1. An unsatisfiability certificate $\mathcal{S}_K(\bot)$ is obtained for F (although F is satisfiable).
2. A checker imports a clause-signature pair (c, s) where c is unsound w.r.t. F but $\mathcal{S}_K(c) = s$, which can lead to incorrect results and hence enable attack 1 (in particular if $c = \emptyset$).
3. A formula $F' \neq F$ results in a collision $\mathcal{S}_K(F) = \mathcal{S}_K(F')$, which allows to re-use signed outputs related to F' for F. This enables attacks 1 (if F' is unsatisfiable) and 2.
4. The secret key K is recovered, which in particular trivially enables attack 1.

All of the outlined attacks involve either to obtain K or to otherwise *forge* a pair $(o, \mathcal{S}_K(o))$ for an object o chosen by the attacker. (For attack 3, finding the desired *second preimage* F' for $\mathcal{S}_K(F)$ implies such a forgery for $o := F'$.) MAC schemes such as SipHash claim protection against both forgery and key recovery. Specifically, if an attacker is able to guess and test 2^x 128-bit SipHash signatures for an object o of their choosing, they can find a valid signature for o with probability 2^{-128+x}. Likewise, if an attacker can guess and test 2^x values of K, they succeed at obtaining K with probability 2^{-128+x} [1]. This presumably holds even if the attacker can query signatures for objects $o' \neq o$ at will. Assuming that a single invalid signature aborts the solving procedure and that a different key K is used for each solving attempt, we have $x = 1$, which yields a success probability of around 10^{-38}. As such, successful forgery or key recovery can be ruled out for all practical purposes.[5]

We argue that the deliberate attacks outlined above cannot be outdone by inadvertent bugs or errors during distributed solving. In particular, data corruption that happens to result in a forged signature must be by pure chance (2^{-128}) as long as K is not accessed.

[5] Schroeder et al. [46] estimate that around 10^4 to 10^5 DRAM errors occur per 10^9 device hours, translating to roughly 10^{-8} memory errors per second. Even if only a byte out of a terabyte (10^{-12}) in a device's main memory was critical to a checker's correctness, a memory error leading to an incorrect result would be a quintillion times more likely ($10^{-8} \cdot 10^{-12} = 10^{-20}$) than 10^{-38}.

Accidental reading of K, in turn, is ruled out since K is available only within the confined address spaces of our trusted modules. For practical purposes, if no protection against malicious intent is required, we consider the use of one fixed K for all solving attempts to be sufficient for highest confidence in the produced results.

3.6 The Road To Verified Distributed Solving

We now briefly discuss how our work relates to potential efforts towards formally verified and, in particular, provably *correct* parallel and distributed SAT solving.

As of yet, neither our on-the-fly checking approach nor its particular implementation are formally verified. We do consider such an undertaking a distinct possibility for future work. In particular, we believe that formal verification efforts for prior LRAT checkers [24, 50] are extensible to the additional interface methods in our trusted modules – under the assumption that our signing scheme guarantees perfect authenticity – with modest effort. Such prior works used verified toolchains like ACL2 or CakeML to produce verified code down to the machine instruction level [24, 50]. Alternatively, we may pursue direct verification of our critical code (see Section 4.1), combined with the use of a verified compiler [31].

Verified SAT solving usually encompasses the aspects of *correctness* (separable into the correctness of SAT and UNSAT results) and *termination* [6]. Verifying the correctness of our approach would equate to obtaining a verified distributed SAT solver in terms of UNSAT correctness. This can easily be extended to SAT correctness by checking the found satisfying assignment in the corresponding checker instance.[6] We consider verification in terms of termination to be of less practical interest for parallel solvers. That being said, verified termination could be achieved by running an isolated verified SAT solver, e.g., IsaSAT [6], next to the portfolio and terminating globally when that solver exits.

4 Evaluation

We now present the evaluation of our approach, beginning with a discussion of our implementation, then outlining our experimental setup and the changes we made to Michaelson et al.'s proof production approach, and finally presenting results. Our experimental data and all our code can be found via `https://github.com/domschrei/mallob-impcheck-data`.

4.1 Implementation

Our implementation bases on MallobSat [44, 45], the state of the art in distributed SAT solving, with LRAT-producing CaDiCaL [39] as a solver backend. We connected CaDiCaL's LRAT proof tracer interface, where each derivation and deletion arrives, to an interface that connects to the solver thread's checker sub-process. The solver thread itself writes each emitted LRAT directive to a single-producer, single-consumer ring buffer R and is then free to continue solving. A dedicated thread polls directives from R and forwards them to the sub-process; another thread reads responses from the sub-process and reacts accordingly, e.g., by exporting a derived clause with the returned signature. As such, a solver thread only needs to wait for its checker sub-process if the solver produces proof information faster than can be processed (in which case the solver thread blocks waiting for free space in R). No unsound derivation nor UNSAT result can ever leave a solver-checker unit; both cases require a valid signature and thus an explicit confirmation from the checker.

[6] We have in fact integrated such a mechanism into our checker interface (see Section 4.1), with the caveat that (at least) one checker per SAT process is required to preserve all original problem clauses, which are needed to check an assignment. Our experimental evaluation does not yet include this feature.

Our trusted modules are written in pure C99, which is well supported for future efforts on verification [4] and verified compilation [31]. We kept this trusted codebase isolated, small, and simple. We only use few standard library features such as memory allocation and file I/O. In terms of program instructions, our inter-process communication via named pipes is indistinguishable from plain, linear file I/O and thus does not expose any private memory segments. Our implementation amounts to around 1000 effective lines of code (ELOC). In comparison, we counted around 40k ELOC in CaDiCaL, 60k ELOC in Mallob [40], and over 300k ELOC in Open MPI (`ompi` and `opal`). As such, our approach may reduce the code critical for a distributed solver's correctness by two orders of magnitude.

We tested our code with small random manipulations during a solving procedure (e.g., inserting a superfluous dependency in a derivation or tampering with a clause literal during clause sharing), confirming that an appropriate error is output and that solving aborts.

4.2 Experimental Setup

We run all experiments on a high-performance cluster called **HoreKa** located in Karlsruhe, Germany. Each used compute node has access to 256 GB of main memory and features two Intel Xeon Platinum 8368 sockets with 38 cores each. Each core consists of two hardware threads, leading to a total of 152 hardware threads per compute node. Nodes can communicate via an InfiniBand 4X HDR interconnect.[7] We made sure that high-bandwidth output, such as proofs, is written to local disks (960 GB NVMe SSD) rather than a network file system.

We consider two solver systems in their latest version: the shared-memory solver Gimsatul [19] and the distributed solver MallobSat. Since we are unsure how well Gimsatul handles 76 cores across two sockets, we run this solver in two modes: on all 76 cores of a node, and on one socket (38 cores) only. Regarding MallobSat, we use 1–32 compute nodes (76–2432 cores) at once, thus covering the range from moderate parallelism up to a massively parallel scale. We spawn two processes per node (one per socket), run up to 38 solvers per process, and leave the remaining hardware threads to other concurrent tasks such as LRAT checking and clause sharing. This is in line with MallobSat's deployment in most earlier works [40, 44, 45], the only difference being that hardware threads not occupied with SAT solving now need to perform more work, which may in turn slow down solver threads.

Regarding MallobSat, we focus on a CaDiCaL portfolio, which proved to perform competitively in recent experiments [36, 45]. We refer to our approach as M-ImpChk ("**M**allobSat with **im**mediate **m**assively **p**arallel **p**ropositional **p**roof **ch**ec**k**ing"), to MallobSat with explicit proof production as M-Proof, and to MallobSat without any proof processing as M-nt ("**n**on **t**rusted"). Our version and configuration of MallobSat is similar to the one described recently [45] apart from the updated CaDiCaL backend (Section 4.1). Since M-ImpChk requires more RAM due to concurrent checking, we halved M-ImpChk's memory threshold per solver, which implies that the largest instances (hundreds of megabytes) are solved with fewer solvers than in M-nt or M-Proof.

We use the 400 benchmark instances from SAT Competition 2023. Note that these benchmarks contain some crafted instances that are designed to be difficult to solve with general resolution [7, 12]. MallobSat can solve some of them when using its Lingeling backend, which features some non-standard reasoning techniques without proof support [5, 27]. Since we use CaDiCaL without any extended resolution features, we ignore the according offset in performance and leave trusted solving with such advanced reasoning for future work.

[7] `https://www.nhr.kit.edu/userdocs/horeka/hardware/`

We allow up to 300 s of wallclock time for solving and up to 1500 s of wallclock time each for combining and for checking a proof. For M-Proof we consider three timings: the *solving time* (ST), which is the time span from program start to a solver reporting a result; the "*time until proof*" (TuP), which is the time span from program start to the presence of a single proof artifact that is checkable in a single linear pass; and the "*time until validation*" (TuV), which corresponds to the TuP plus the time required to check the proof. For M-ImpChk, there is no TuP; its TuV corresponds to its ST, since we stop the ST at the point where an unsatisfiability result is *confirmed* by a checker instance. We primarily assess an approach in terms of its ST / TuP / TuV *overhead*, which is the ratio between its ST / TuP / TuV and the ST of M-nt, minus one.

4.2.1 Updates to Proof Production

We now outline the changes we made to M-Proof in order to make its running times more competitive and to render proof production and checking more feasible in our experiments. Note that these changes will shortly be published separately and in more detail [37].

The updated CaDiCaL backend allowed us to remove a previously required sequential preprocessing step that exhaustively performed unit propagation on the input. We correspondingly do not need to produce and prepend a proof for this preprocessing. Moreover, the original setup featured postprocessing where the inverted combined proof is un-inverted and syntactically transformed to feature a compact domain of IDs. This step was a necessity because of relatively poor tool support for the kind of LRAT proofs M-Proof emits (albeit perfectly valid in principle). Specifically, `lrat-check` from the `drat-trim` toolbox is not able to gracefully handle large gaps between subsequent clause IDs. The more recent checker `lrat-trim` [39] operates on 32-bit IDs and is hence not suitable either. In our setup, we use a standalone, fast LRAT checker crafted from MallobSat code that can operate directly on the compressed and inverted proof. As such, we measure the TuP up to the point where the combined, compressed, and reversed proof has been written. The TuV then adds a single linear read of the proof file,[8] thus minimizing I/O operations.

4.3 Results

We first compare the solving times (ST) of all considered approaches. Fig. 4 provides an overview, with exact values provided in Tab. 1. Note again that this does *not* include the time required by M-Proof to combine and check proofs nor the DRAT proof checking time for Gimsatul. All approaches consistently show improved performance when increasing the computational resources. Gimsatul did not reach the performance of MallobSat. As a side note, we consider it notable that running Gimsatul across both sockets does perform better than only using a single socket. M-Proof now performs very similarly to M-nt following our updates (*cf.* [36]). The overhead of M-ImpChk, which we analyze below, is more noticeable. Still, M-ImpChk at 16 (4) nodes is able to outperform all approaches at 4 (1) nodes, and M-ImpChk at 32 nodes is on par with M-nt at 16 nodes.

Fig. 5 provides a detailed comparison of M-nt vs. M-ImpChk solving times at the smallest and largest scale considered. The incurred overhead is consistent, stable, and does not correlate with M-nt's solving times (Pearson's r: -0.11 at 1 node, -0.16 at 32 nodes).

Fig. 6 (leftmost plot) shows the per-instance overhead of M-ImpChk. At all scales, the observed overhead is below 70% (i.e., slowdown 1.7) for three out of four instances and at most 42% for the median instance. Specifically, the median overhead ranges from 37.1%

[8] Our reverse file parser uses buffering to reverse-read roughly as fast as usual forward reading.

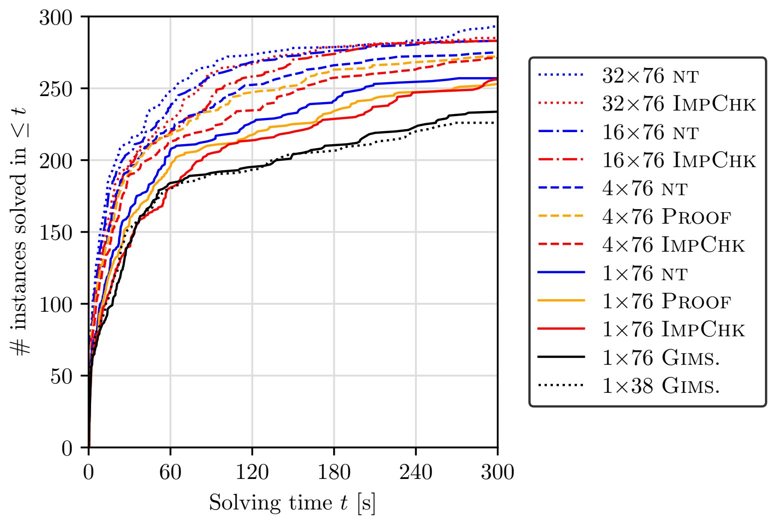


Figure 4 Pure solving times of M-NT, M-PROOF (*excluding proof combination and checking*), M-IMPCHK, and GIMSATUL (*excluding proof checking*), on 1–32 nodes. Colors delineate the approaches, line styles indicate the scale of solving. "16×76 PROOF" is omitted to reduce clutter; the corresponding curve is closely aligned with "16×76 NT".

(at four nodes) to 42.1% (at 32 nodes), with the geometric mean ranging from 37.7% (at 16 nodes) to 45.6% (at one node). In addition, we averaged the checker processes' CPU utilization during a solving attempt to analyze how much time the checkers spend on actual processing vs. waiting for the next directive. We arrived at a median utilization from 11.2% at one node to 12.4% at 32 nodes (geometric mean 10.4%–11.0%), which indicates that our checkers are mostly idle[9] and, in turn, have no difficulties with processing the produced proof information and computing signatures on-the-fly. The increased communication volume

[9] The low CPU ratio is not a consequence of over-subscribed hardware threads. All non-solver, non-checker threads use little CPU time and thus leave plenty of CPU time for the checkers.

Table 1 Solved instances ("#") and PAR-2 scores ("PAR") of each run, only considering pure solving times (*excluding proof combination/checking* for M-PROOF and GIMSATUL).

		all		satisf.		unsat.	
	# nodes	#	PAR	#	PAR	#	PAR
M-NT	1	257	241.5	120	248.7	137	234.3
	4	275	211.5	124	233.9	151	189.1
	16	283	196.2	126	229.2	157	163.3
	32	293	184.2	128	222.6	165	145.7
M-PROOF	1	253	250.0	119	254.1	134	245.9
	4	272	216.7	122	239.1	150	194.3
	16	282	197.4	126	228.5	156	166.4
M-IMPCHK	1	257	251.5	122	250.2	135	252.7
	4	271	222.3	124	235.0	147	209.6
	16	283	201.0	128	225.8	155	176.3
	32	285	195.3	129	221.5	156	169.1
GIMSATUL	1/2	226	286.3	120	250.5	106	322.1
	1	234	279.1	122	246.7	112	311.6

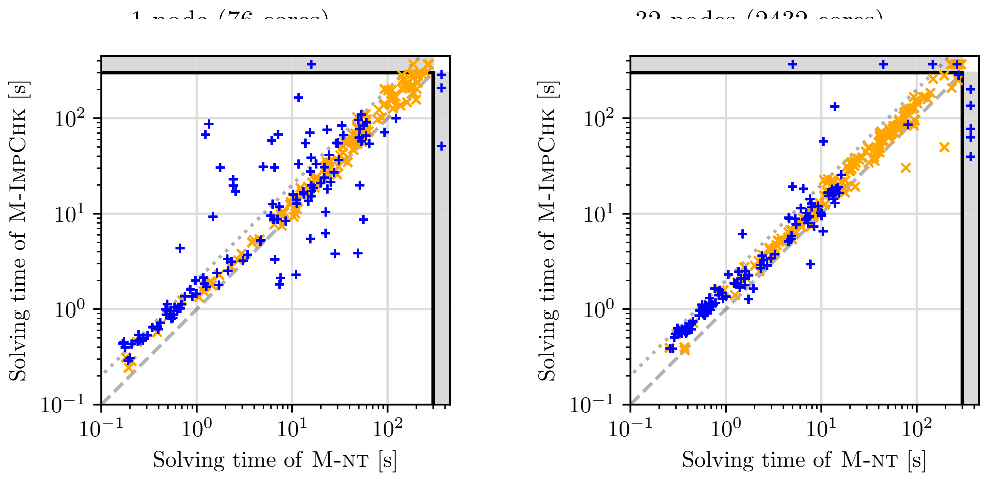


Figure 5 Solving times of M-NT vs. M-ImpChk on one (left) and 32 nodes (right). Points on the central diagonal denote instances solved equally fast; the dotted diagonal shows where M-ImpChk takes twice as long. Blue "+" denote satisfiable, orange "×" unsatisfiable instances.

(with each clause now carrying 192 bits of metadata) appears to be unproblematic as well, which is unsurprising considering that MallobSat does not even remotely exhaust the communication bandwidth of HPC interconnects [45]. All of these observations support that our approach to trusted solving is scalable and bottleneck-free.

Fig. 6 (right) shows the overhead incurred by M-Proof. The median ST overhead is below 11% at all scales, which indicates that producing and writing LRAT proof information is inexpensive in and of itself. The TuP overhead, by contrast, is substantial and increases with the scale of solving; even at a single node (median 104%), it surpasses the TuV overhead of our approach at *any* scale. Going from 4 to 16 nodes, the growing TuP overhead (median 196% to 275%) even causes a decreasing number of successfully produced proofs (146 to 141).

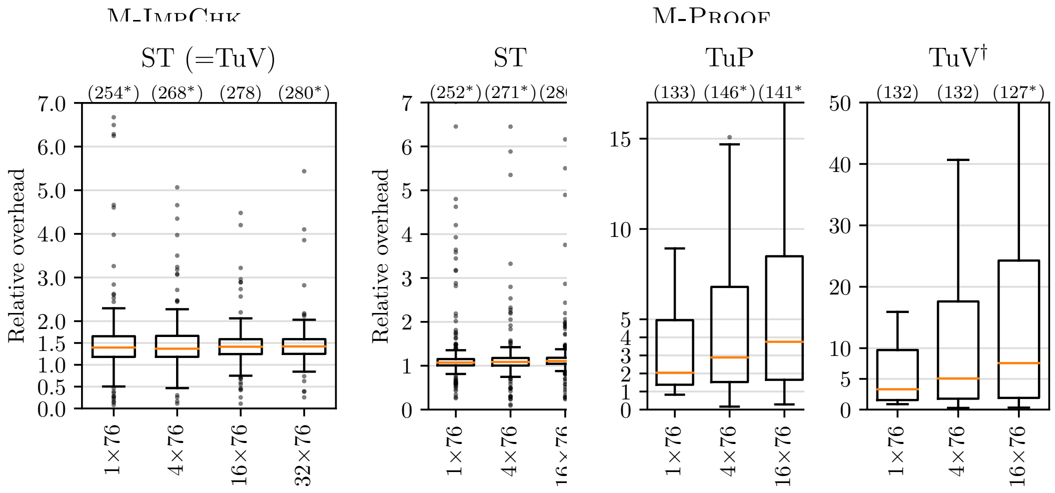


Figure 6 ST overhead (= TuV overhead) of M-ImpChk; and ST, TuP, and TuV overhead (the latter two for unsatisfiable instances only) of M-Proof. Numbers at the top show the number of considered data points. *Some data surpasses the displayed interval and has been cut off. †TuV overheads at 4 and 16 nodes are not real measurements but estimated based on the checker's proof traversal time at one node scaled by the obtained proof size at 4 and 16 nodes respectively.

Across the 127 instances where M-Proof produced a proof at all three tested scales, the mean proof size increases from 7.33 GiB at one node to 10.73 GiB (+46%) at four nodes and finally to 14.46 GiB (another +35%) at 16 nodes. As such, it became increasingly challenging to store and/or check all produced proofs, which exceed 4.5 TiB for the 16-node run alone. Therefore, for all runs beyond a single node, we only retained the size (in bytes) of a proof and then deleted it without checking. To still gain an impression on the TuV at 4 and 16 nodes, we extrapolated estimates based on the assumption that, for a fixed instance, a proof's checking time grows linear in the proof's file size. We estimated an instance's TuV at 4 (16) nodes by its TuP plus its one-node proof checking time, scaling the time needed to traverse the proof by the obtained proof size at 4 (16) nodes. This extrapolation only covers instances where already the single-node run produced and checked a proof, and the estimates can exceed our actual proof checking time limit of 1500 s. The estimated median TuV overhead increases steeply with the scale of solving – from 233% at one node to 416% at four nodes and to 657% at 16 nodes. Overall, our experiments and analysis have fully confirmed our concerns with respect to the scalability of M-Proof.

The complete DRAT proofs output by Gimsatul average 2.36 GiB for the 38-core variant and 3.10 GiB for the 76-core variant. Out of 97 38-core proofs that `drat-trim` deemed complete and sound, only 65 proofs were checked successfully, at a median overhead of 475% (geometric mean 629%), while 32 checking attempts timed out. This confirms that checking DRAT proofs from parallel solvers can at times border on infeasible [19, 26].

Lastly, we tracked MallobSat's peak global RAM usage (measured once a second). For the median instance at 32 nodes, M-ImpChk increased the RAM usage over M-nt by around 60%. Note that we reduced M-ImpChk's per-solver memory threshold (Section 4.2), which however was only triggered in four cases. By contrast, M-Proof incurs virtually no RAM overhead (median $< 3\%$ at all scales) since all proof information is written to disk and processed with external-memory data structures [43, Sect. 5.5].

5 Discussion

In the following, we discuss the merits and the limitations of the proposed approach.

We believe that our approach can be applied to many other clause-sharing solvers. Its requirements are that each solver thread outputs LRAT information and that each clause is shared with its LRAT ID and its signature. Unlike M-Proof, whose proof combination is based on periodic all-to-all clause sharing, our checking can be used with any clause sharing, including exchanges along rings [54] or communication graphs [17]. On a pragmatic level, we release our trusted modules (Section 4.1) together with introductory examples and documentation on how developers can integrate them into a parallel solver.

Our approach does not rely on I/O speed nor large amounts of disk space. While not yet implemented, it is also entirely possible to perform on-the-fly checking while additionally writing partial proofs to later serve as witnesses. Moreover, on-the-fly checking easily allows for *malleable* setups, where solvers may be added or removed during solving. This is useful for scheduling and solving many SAT instances at once [40] and for reacting to main memory shortages [45]. In addition, we anticipate our approach to be useful for solver development. Pollitt et al. found on-the-fly checking to be "*dramatically reducing the implementation effort (particularly for debugging)*" [39], which is now possible at a distributed level.

A limitation of our approach in its current form is its relatively high RAM usage. While the asymptotic memory usage of solving remains the same, an increase by a constant factor is to be expected since we now maintain *two* clause databases per solver thread – one within the solver and one in the LRAT checker. A possible measure to make RAM usage less of an issue would be to let the solver threads within a process share data structures, as in Gimsatul [19]. Similarly, our t checker sub-processes per SAT process could be combined into a *single* checker process. This process may need to be multi-threaded in order to achieve the required throughput. Still, memory usage could be reduced significantly by storing equivalent clauses only once while reference-counting them for safe deletion (cf. [26]). To further reduce our approach's memory footprint, it may also be worthwhile to compress the checkers' clause databases in some inexpensive manner (*cf.* [38]).

All in all, our impression is that, in a few years, on-the-fly proof checking may be sufficiently advanced and widely supported that the International SAT Competition is in a position to require verified results across all principal tracks (sequential, parallel, cloud). We would consider this a significant advancement in terms of reliable and trustworthy SAT solving that can be used even for critical matters in a carefree manner.

6 Conclusion

Motivated by the insufficient scalability of producing proofs in distributed systems, we have presented a novel approach to trusted distributed clause-sharing solving where LRAT information is checked on-the-fly. The critical code is small, simple, and kept separate from solver code, and the solver's clause sharing does not need to be trusted. We confirmed our approach to be bottleneck-free and to scale drastically better than prior trusted approaches with explicit proof combination. To the best of our knowledge, this is the first instance of a trustworthy parallel solver whose running times are dominated by solving, not checking.

Regarding future work, we aim to formally verify our approach in order to obtain a provably correct SAT solver (see Section 3.6). In addition, we are interested in adding LRAT support to further solvers, in particular Gimsatul and Kissat, to make our approach more broadly applicable, boost its performance, and reduce its main memory requirements. Lastly, we intend to add full LRAT (rather than LRUP) checking to our approach, which may help with integrating solvers with advanced reasoning into parallel and distributed solvers.

References

1 Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012. doi:10.1007/978-3-642-34931-7_28.

2 Tomáš Balyo and Carsten Sinz. Parallel satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*. Springer, 2018. doi:10.1007/978-3-319-63516-3_1.

3 Anton Belov, Marijn J. H. Heule, and João Marques-Silva. MUS extraction using clausal proofs. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 8561, pages 48–57. Springer, Springer, 2014. doi:10.1007/978-3-319-09284-3_5.

4 Dirk Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 133–155. Springer, 2019. doi:10.1007/978-3-030-17502-3_9.

5 Armin Biere. Lingeling, Plingeling and Treengeling entering the SAT competition 2013. In *SAT Competition*, volume 2013, page 1, 2013. URL: https://fmv.jku.at/papers/Biere-SAT-Competition-2013-Lingeling.pdf.

6 Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of automated reasoning*, 61:333–365, 2018. doi:10.1007/s10817-018-9455-7.

7 Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. Crafted benchmark formulas requiring symmetry breaking and/or parity reasoning. In *SAT Competition*, page 67, 2023. URL: https://researchportal.helsinki.fi/files/269128852/sc2023_proceedings.pdf.

8 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 44–57. Springer, 2010. doi:10.1007/978-3-642-14186-7_6.

9 Mark Alexander Burgess, Charles Gretton, Josh Milthorpe, Luke Croak, Thomas Willingham, and Alwen Tiu. Dagster: Parallel structured search with case studies. In *Pacific Rim Int. Conf. Artificial Intelligence*, pages 75–89. Springer, 2022. doi:10.1007/978-3-031-20862-1_6.

10 Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In *ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 11–20, 2020. doi:10.1145/3377813.3381347.

11 Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19:7–34, 2001. doi:10.1023/A:1011276507260.

12 Cayden R. Codel, Joseph E. Reeves, and Randal E. Bryant. Pigeon hole and mutilated chessboard with mixed constraint encodings and symmetry-breaking. In *SAT Competition*, page 72, 2023. URL: https://researchportal.helsinki.fi/files/269128852/sc2023_proceedings.pdf.

13 Byron Cook. Automated reasoning's scientific frontiers. https://www.amazon.science/blog/automated-reasonings-scientific-frontiers, 2021. Amazon Science.

14 Luis Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *Automated Deduction – CADE 26*, pages 220–236, 2017. doi:10.1007/978-3-319-63046-5_14.

15 Adnan Darwiche and Knot Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability*, pages 101–132. IOS Press, 2021. doi:10.3233/faia200987.

16 Daniel DeFreez, Antara Bhowmick, Ignacio Laguna, and Cindy Rubio-González. Detecting and reproducing error-code propagation bugs in MPI implementations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–201, 2020. doi:10.1145/3332466.3374515.

17 Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. Communication in massively-parallel SAT solving. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 709–716. IEEE, 2014. doi:10.1109/ictai.2014.111.

18 Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Communications of the ACM*, 66(6):64–72, 2023. doi:10.1145/3560469.

19 Mathias Fleury and Armin Biere. Scalable proof producing multi-threaded SAT solving with Gimsatul through sharing instead of copying clauses. In *Pragmatics of SAT*, 2022.

20 Nils Froleyks, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

21 Edgar N. Gilbert, F. Jessie MacWilliams, and Neil J. A. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974. doi:10.1002/j.1538-7305.1974.tb02751.x.

22 Marijn J. H. Heule. The DRAT format and DRAT-trim checker. *CoRR*, abs/1610.06229, 2016. doi:10.48550/arXiv.1610.06229.

23 Marijn J. H. Heule, Warren Hunt, and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013. doi:10.1109/fmcad.2013.6679408.

24 Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *International Conference on Interactive Theorem Proving*, pages 269–284, 2017. doi:10.1007/978-3-319-66107-0_18.

25 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011. doi:10.1007/978-3-642-34188-5_8.

26 Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. Validating unsatisfiability results of clause sharing parallel SAT solvers. In *Pragmatics of SAT*, pages 12–25, 2014. doi:10.29007/6vwg.

27 Alexey Ignatiev, António Morgado, and João Marques-Silva. On tackling the limits of resolution in SAT solving. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 164–183. Springer, 2017. doi:10.1007/978-3-319-66263-3_11.

28 George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *AAAI Conference on Artificial Intelligence*, volume 27, pages 481–488, 2013. doi:10.1609/aaai.v27i1.8660.

29 Benjamin Kiesl, Adrián Rebola-Pardo, and Marijn JH Heule. Extended resolution simulates DRAT. In *International Joint Conference on Automated Reasoning*, pages 516–531. Springer, 2018. doi:10.1007/978-3-319-94205-6_34.

30 Peter Lammich. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning*, 64(3):513–532, 2020. doi:10.1007/s10817-019-09525-z.

31 Xavier Leroy. The CompCert C verified compiler: Documentation and user's manual. Technical report, Inria, 2023. URL: https://inria.hal.science/hal-01091802v11.

32 Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. Verification of MPI programs using CIVL. In *24th European MPI Users' Group Meeting*, pages 1–11, 2017. doi:10.1145/3127024.3127032.

33 Norbert Manthey, Tobias Philipp, and Christoph Wernhard. Soundness of inprocessing in clause sharing SAT solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 22–39. Springer, 2013. doi:10.1007/978-3-642-39071-5_4.

34 João Marques-Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *IEEE International Symposium on Fault Tolerant Computing*, pages 152–161. IEEE, 1997. doi:10.1109/ftcs.1997.614088.

35 João Marques-Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *Annual Design Automation Conference (DAC)*, pages 675–680, 2000. doi:10.1145/337292.337611.

36 Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Unsatisfiability proofs for distributed clause-sharing SAT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 348–366. Springer, 2023. doi:10.1007/978-3-031-30823-9_18.

37 Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. Producing proofs of unsatisfiability with distributed clause-sharing SAT solvers. *Journal of Automated Reasoning*, 2024. In preparation.

38 Alexander Nadel. Solving huge instances with Intel(R) SAT solver. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.17.

39 Florian Pollitt, Mathias Fleury, and Armin Biere. Faster LRAT checking than solving with CaDiCaL. In *Theory and Applications of Satisfiability Testing (SAT)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.21.

40 Peter Sanders and Dominik Schreiber. Decentralized online scheduling of malleable NP-hard jobs. In *European Conference on Parallel Processing (Euro-Par)*, pages 119–135. Springer, 2022. doi:10.1007/978-3-031-12597-3_8.

41 André Schidler and Stefan Szeider. SAT-based decision tree learning for large data sets. In *AAAI Conference on Artificial Intelligence*, volume 35, pages 3904–3912, 2021. doi:10.1609/aaai.v35i5.16509.

42 Dominik Schreiber. Lilotane: A lifted SAT-based approach to hierarchical planning. *Journal of Artificial Intelligence Research (JAIR)*, 70:1117–1181, 2021. doi:10.1613/jair.1.12520.

43 Dominik Schreiber. *Scalable SAT Solving and its Application*. PhD thesis, Karlsruhe Institute of Technology, 2023. doi:10.5445/IR/1000165224.

44 Dominik Schreiber and Peter Sanders. Scalable SAT solving in the cloud. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 518–534. Springer, 2021. doi:10.1007/978-3-030-80223-3_35.

45 Dominik Schreiber and Peter Sanders. MallobSat: Scalable SAT solving by clause sharing. *Journal of Artificial Intelligence Research (JAIR)*, 2024. In press.

46 Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009. doi:10.1145/2492101.1555372.

47 Laurent Simon. Post mortem analysis of SAT solver proofs. In *Pragmatics of SAT*, pages 26–40, 2014. doi:10.29007/gpp8.

48 Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. PaSAT – parallel SAT-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001. doi:10.1016/s1571-0653(04)00323-3.

49 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 244–257. Springer, 2009. doi:10.1007/978-3-642-02777-2_24.

50 Yong Kiam Tan, Marijn J. H. Heule, and Magnus Myreen. Verified LRAT and LPR proof checking with cake_lpr. In *SAT Competition*, page 89, 2023. URL: https://researchportal.helsinki.fi/files/269128852/sc2023_proceedings.pdf.

51 Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. *ACM Sigplan Notices*, 44(4):261–270, 2009. doi:10.1145/1594835.1504214.

52 Sean Weaver and Marijn J. H. Heule. Constructing minimal perfect hash functions using SAT technology. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 1668–1675, 2020. doi:10.1609/aaai.v34i02.5529.

53 Lihao Xu and Cheng Huang. Computation-efficient multicast key distribution. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):577–587, 2008. doi:10.1109/TPDS.2007.70759.

54 Xindi Zhang, Zhihan Chen, and Shaowei Cai. PRS: A new parallel/distributed framework for SAT. In *SAT Competition*, pages 39–40, 2023. URL: https://researchportal.helsinki.fi/files/269128852/sc2023_proceedings.pdf.

Optimal Layout Synthesis for Deep Quantum Circuits on NISQ Processors with 100+ Qubits

Irfansha Shaik
Department of Computer Science, Aarhus University, Denmark
Kvantify Aps, Copenhagen S, Denmark

Jaco van de Pol
Department of Computer Science, Aarhus University, Denmark

Abstract

Layout synthesis is mapping a quantum circuit to a quantum processor. SWAP gate insertions are needed for scheduling 2-qubit gates only on connected physical qubits. With the ever-increasing number of qubits in NISQ processors, scalable layout synthesis is of utmost importance. With large optimality gaps observed in heuristic approaches, scalable exact methods are needed. While recent exact and near-optimal approaches scale to moderate circuits, large deep circuits are still out of scope. In this work, we propose a SAT encoding based on parallel plans that apply 1 SWAP and a group of CNOTs at each time step. Using domain-specific information, we maintain optimality in parallel plans while scaling to large and deep circuits. From our results, we show the scalability of our approach which significantly outperforms leading exact and near-optimal approaches (up to 100x). For the first time, we can optimally map several 8, 14, and 16 qubit circuits onto 54, 80, and 127 qubit platforms with up to 17 SWAPs. While adding optimal SWAPs, we also report near-optimal depth in our mapped circuits.

2012 ACM Subject Classification Hardware → Quantum computation; Computing methodologies → Planning for deterministic actions

Keywords and phrases Layout Synthesis, Transpiling, Qubit Mapping and Routing, Quantum Circuits, Propositional Satisfiability, Parallel Plans

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.26

Related Version *Previous Version*: `https://arxiv.org/abs/2403.11598`

Supplementary Material *Software*: `https://github.com/irfansha/Q-Synth` [24]
archived at `swh:1:dir:be31c57364bd541c6b65afac80603e9004cf4008`
Software (Github Tag): `https://github.com/irfansha/Q-Synth/releases/tag/Q-Synth-v2.0-SAT2024` [25], archived at `swh:1:dir:be31c57364bd541c6b65afac80603e9004cf4008`

Funding IFD (Innovation Fund Denmark)

1 Introduction

The Quantum Layout Mapping problem takes as input a quantum circuit (logical design) and a coupling map (connectedness between physical qubits). The result is an "equivalent" quantum circuit mapped to the physical qubits, such that any binary operation only happens on connected qubits. Besides an initial mapping of logical qubits to physical qubits, this also involves the insertion of SWAP gates. Noise is inherent to qubits in Noisy Intermediate-Scale Quantum (NISQ) processors. Additional SWAP gates increase both the 2-qubit gate count and the circuit depth. In the current NISQ era, minimizing error is of utmost importance for any practical quantum computing. The error rate depends on the number of gates, the fidelity of gates, and the depth of the circuit. The Optimal Quantum Layout Synthesis is to synthesize a mapping that optimizes one of the above metrics.

Optimal Layout Synthesis has been studied before. A nice overview is provided in [29]. Several heuristic approaches exist which optimize various metrics. The classical algorithm for heuristic mapping is SABRE (in Qiskit) [15]. [27] use the MQT benchmarks for mapping and swapping, using a heuristic search space reduction with an $O(n \log n)$ algorithm. Other approaches used include A* with cost metrics [32], MAXSAT [19], temporal planning [30], and constraint programming [3] (minimizing circuit depth).

While heuristic approaches are fast and scalable, their suboptimal mappings may result in high error rates [31, 29]. Optimizing fidelity with exact approaches can result in circuits with the lowest error rate. However, as shown in [28], optimizing fidelity is extremely hard and does not scale beyond small circuits. Circuit depth and 2-qubit gate count optimization are better alternatives for scalability. The OLSQ tool[1] optimizes circuit depth and is built on [28]. A scalable variant OLSQ2 based on Z3 appeared in [16]. The QMAP tool[2] optimizes the number of SWAP gates and is based on [33, 31]. The same authors introduced the use of subarchitectures [22]. Other ideas to improve quantum layout use quantum teleportation [10]. In [4], measurements are placed early so qubits can be reused.

In [26], we proposed a tool, Q-Synth v1[3], for SWAP gate optimization which outperformed both QMAP and OLSQ tools. Q-Synth v1 reduces optimal quantum layout synthesis to classical planning. For maintaining the optimality of the SWAP gates added, Q-Synth v1 adds exactly 1 CNOT or 1 SWAP gate at each time step. In such an approach, the hardness increases with the plan length i.e., the number of CNOTs + SWAPs. Despite the recent progress in Q-Synth v1 and OLSQ2, deep circuits that require many SWAPs are still out of reach.

Contribution

In this paper, we provide a SAT encoding based on parallel plans with domain-specific information. In particular, at each time step, we map one SWAP gate and a group of CNOT gates. This reduces the make-span, and using domain-specific information we maintain the optimality. We propose two-way constraints for CNOT dependencies for better dependency propagation. In addition, we also provide variations of our encoding with bridges and relaxed dependencies (via commutation). In all variations, we only add provably optimal number of bridges+SWAPs.

For experimental evaluation, we consider two benchmark sets: 1) Standard benchmarks from previous papers; and 2) Deep VQE benchmarks. For comparison, we consider leading near-optimal tool TB-OLSQ2 [16] and heuristic SABRE [15]. For mapping, we consider 4 NISQ processors Melbourne (14 qubits), Sycamore (54 qubits), Rigetti (80 qubits), and Eagle (127 qubits). We propose three experiments: in the first two experiments we map both benchmark sets to the Sycamore, Rigetti, and Eagle platforms. In the first experiment, we compare the number of SWAPs added by all three tools. In the second experiment, we compare SWAP additions and circuit depth of the mapped circuits with TB-OLSQ2. In the third experiment, we compare the effectiveness of bridges and relaxed dependencies in our tool by mapping onto the Melbourne platform. Here we report the additional number of (optimal) SWAPs+bridges.

We demonstrate that our encoding can optimally map deep circuits onto large platforms with up to 127 qubits. Our tool outperforms the leading near-optimal tool TB-OLSQ2 up to 100x while always adding the optimal number of SWAPs. We show that while adding optimal SWAPs, we also report near-optimal depth in the mapped circuits. We also confirm that heuristic approaches like SABRE add too many SWAPs.

[1] OLSQ tool `https://github.com/tbcdebug/OLSQ`
[2] Munich Quantum Toolkit QMAP `https://github.com/cda-tum/qmap`
[3] Q-Synth v1 tool `https://github.com/irfansha/Q-Synth/releases/tag/Q-Synth-v1.0-ICCAD23`

2 Preliminaries

2.1 Layout Synthesis for Quantum Circuits

A quantum circuit consists of a fixed number of (logical) qubits, and a number of quantum gates (operations) that are applied to some qubits in a particular order. If the output qubit of gate g_1 is used as an input qubit of gate g_2, we say g_2 depends on g_1. The dependencies form a DAG (directed acyclic graph) between the gates. Gates that are (transitively) independent are called parallel, and can be applied in any order.

Any quantum circuit can be decomposed to an intermediate representation with only single-qubit gates and CNOT gates [7]. Viewed classically, the binary CNOT gate (controlled-NOT, also known as CX) takes two qubits (a, b) as input and transforms them into $(a, a \oplus b)$, i.e., the control qubit a determines whether the data qubit b is negated. We will also use the SWAP gate, which transforms a qubit pair (a, b) into (b, a). A SWAP gate can be expressed as a sequence of 3 CNOT gates.

The single-qubit gates will be treated as black-boxes in this paper; they are distinguished by their name (X, Z, H, S, T, etc.) but we don't make assumptions on their semantics (except in an extension of our method in Section 3.1). We refer the interested reader to [21] for a detailed introduction to quantum gates and quantum circuits in general.

Most physical quantum platforms have limited connectivity, in which the CNOT operations can only be applied on physical qubits that are neighbors in the so-called coupling graph. Given such a circuit and a coupling graph, Layout Synthesis consists of two phases: Initial Mapping and Qubit Routing. In Initial Mapping, the logical qubits of the given circuit are mapped to some physical qubits of the platform bijectively. In Qubit Routing, the following constraints must be satisfied:

- Every gate must be scheduled in an order that respects all dependencies;
- Every gate must be applied to the correct qubits (taken the mapping into account);
- The 2-qubit CNOT gates can only be mapped on connected physical qubits.

Additional SWAP gates may be required, to swap the values of connected physical qubits to ensure all CNOT gates can be mapped. In this paper, we use gate count as an optimization metric. In Layout Synthesis, the number of single-qubit gates and CNOT gates remain unchanged. Optimal Layout Synthesis is thus minimizing the additional SWAP gates.

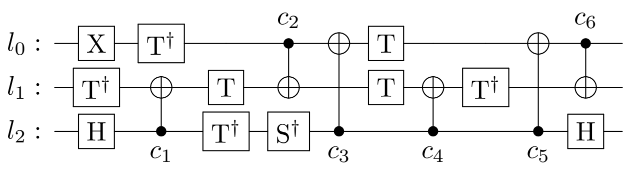
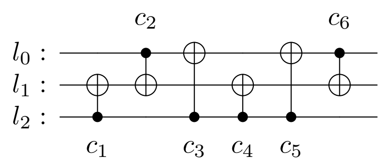
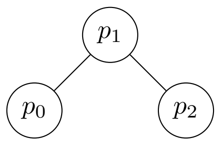


Figure 1 3-qubit Or circuit with 6 CNOT gates.

For example, Figure 1 shows an Or-circuit with 3 logical qubits (horizontal lines $\{l_0, l_1, l_2\}$). The circuit has 11 single-qubit gates (boxes with names) and 6 CNOT gates (the dot indicates the control qubit, while the $\oplus$ indicates the data qubit). Let us suppose we want to map this circuit onto the linear 3-qubit platform as in Figure 2b. Regardless of physical qubit connections, single-qubit gates can always be scheduled. Only the 2-qubit CNOT gates are relevant for our optimal synthesis problem. Thus, we first remove the single-qubit gates and only consider CNOT gates for the mapping. After finding the optimal mapping, the single-qubit gates will be reinserted. Figure 2a shows the CNOT gates in the Or-circuit.



(a) Or-circuit with only CNOT gates.



(b) Coupling graph.

Figure 2 Reduced Or circuit and a 3-qubit linear platform.

In any valid mapping, the dependencies must be respected, for example, gates c_1 and c_3 can only be mapped before and after gate c_2, respectively. In this example, the dependency graph is a total order, but note that with 4 qubits, parallel CNOT gates are possible, which can be scheduled in any order. One can observe that the connections of the CNOT gates c_1, c_2 and c_3 form a triangle $(l_0, l_1), (l_1, l_2), (l_2, l_0)$. Since the coupling graph does not have a triangle, one cannot map our example circuit to the linear platform. At least two SWAP gates are needed for any valid mapping.

Figure 3 shows such a mapped Or-circuit where l_0, l_1, l_2 are mapped to p_0, p_1, p_2 respectively. Intuitively, the SWAP gates slice the circuit such that the sub-circuits do not have any triangles by CNOT connections.

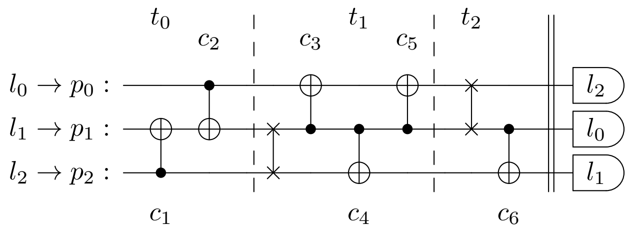


Figure 3 Mapped Or-circuit with 2 additional SWAPs (optimal).

Finally, single-qubit gates can be inserted back respecting original DAG dependencies. Figure 4 shows the final mapped circuit with optimal SWAP gates. Note that the number of physical qubits can be more than logical qubits. In such cases, one can use so-called *ancillary* qubits to avoid unnecessary swaps. Similar to Q-Synth v1, we allow ancillary swapping i.e., a mapped physical qubit can be swapped with an empty physical qubit.

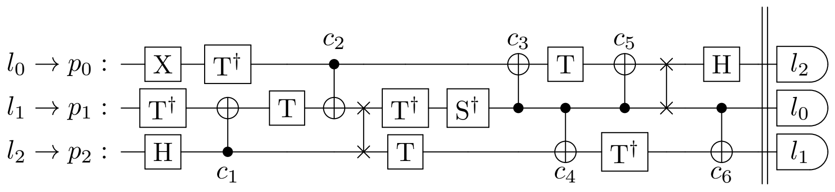


Figure 4 Final Mapped Or-circuit after inserting back single-qubit gates.

2.2 Optimal Layout Synthesis as Planning

In Q-Synth v1, we encoded optimal layout synthesis as a planning problem in the Planning Domain Definition (PDDL) Specification. As discussed above, the reduced circuit with only CNOT gates is mapped using additional SWAPs. Later single-qubit gates are inserted back to reconstruct the final mapped circuit. In such a planning problem, either exactly one CNOT gate or one SWAP gate is scheduled at each time step. A plan with the optimal number of actions corresponds to the optimal number of SWAP additions.

Planning as SAT

Given a boolean formula, a Satisfiability (SAT) problem is finding an assignment to the boolean variables that makes it a true formula. A planning problem can be encoded as a bounded reachability problem. Sequential encoding [14] is a standard SAT encoding where each time step encodes a single action. Using a sequential encoding, one can obtain optimal plans by incrementing the plan length by 1. For instance, one could use Q-Synth v1 with Madagascar (a SAT-based planner) to find an optimal mapping. Since the optimal plan length for the example, Or-circuit is 8 (6 CNOTs + 2 SWAPs), the SAT instance has a make-span of 8. As shown in [26], sequential encoding scales well for moderate circuits however deep circuits are still out of reach. It is consensus that a long optimal plan length can severely impact the performance of SAT-based planners.

Parallel Plans

In literature, alternative parallel plan encodings like ∀-step [13] and ∃-step [23] were proposed for scaling heuristic SAT-based planning. The key idea in a parallel plan is to group two or more actions whose preconditions and effects do not conflict. While encodings like ∃-step scale well, the optimality is not guaranteed. For a scalable optimal layout synthesis, one needs a way to group CNOTs while still maintaining the optimality.

2.3 Parallel Plans in Optimal Layout Synthesis

Madagascar implements both ∀-step and ∃-step parallel plans. Directly using Q-Synth v1 with parallel plan encodings in Madagascar does not preserve optimality. In particular, there are three main challenges:

- More than one SWAP gate can be applied at each parallel step;
- Planner needs to find a partial order in each parallel step satisfying dependencies;
- Relaxing CNOT dependencies within a parallel step is not trivial in a PDDL specification.

In this paper, we directly encode Layout Synthesis as a SAT problem to circumvent the encoding challenges in a PDDL specification. In our encoding, we allow exactly one SWAP gate at each parallel step. Thus, the number of parallel steps corresponds to the optimal SWAP additions. Further, we use domain-specific information from Layout Synthesis to relax CNOT dependencies within a parallel step. In particular, we make two observations:

- The qubit mappings do not change between two consecutive SWAP gates;
- Given a set of CNOTs, one can always reconstruct a partial order with DAG dependencies.

We take advantage of the partial order reconstruction and drop dependency constraints within a parallel step. The SAT solver can now choose exactly one SWAP and a group of CNOTs in each parallel step. CNOT gates in different time steps still need to respect the original DAG dependencies. The full SAT encoding is later discussed in detail in Section 3.

For our example, a parallel plan with a make-span of 3 is sufficient (see Figure 3) instead of 8. At time step 0, along with the initial mapping, a group of CNOT gates are also mapped. From time step 1, exactly one SWAP gate and a group of CNOT gates are mapped. Note that the satisfying assignment returned by a SAT solver only specifies that:

- Logical qubits l_0, l_1, l_2 are mapped to p_0, p_1, p_2, respectively;
- SWAP gates on p_1, p_2 and p_0, p_1 are applied at time steps t_1, t_2 respectively;

- c_1, c_2 gates are applied at t_0;
- c_3, c_4, c_5 gates are applied at t_1;
- c_6 gate is applied at t_2;

In mapped circuit reconstruction, we use the DAG dependencies to order the group of CNOTs in each time step. In the literature, an SMT based encoding is applied in TB-OLSQ(2) [28, 16] which also groups CNOT gates between consecutive SWAP gates. They optimize make-span of their defined problem. However, longer make-span may result in better SWAP count and circuit depth. In our experiments, we indeed observe suboptimal solutions by TB-OLSQ2 in both metrics.

2.4 Incremental SAT Solving

Conflict Driven Clause Learning (CDCL) [18] is a key part of state-of-the-art SAT solving. When solving similar instances, one can reuse the learned clauses. Incremental SAT solving allows solving a SAT instance given an assumption of a partial assignment. Essentially by using different assumptions, multiple instances can be solved while reusing the learned clauses. In problems like planning, one needs to refute up to k-1 plan length for optimal plans. By adding assumptions encoding that the goal is reached in the current iteration, one can solve a planning instance incrementally.

3 Two-Way Parallel SAT encoding

In this section, we implement the ideas discussed above. We provide an incremental SAT encoding that applies the idea of parallel plans in Layout Synthesis. Table 1 describes the main variables used in the encoding. Algorithm 1 describes the structure of our encoding. In every time step, a group of CNOTs are applied. From time step 1, each incremental step adds one extra SWAP. We generate a set of variables for CNOT and SWAP constraints at each time step.

Table 1 Encoding variables and descriptions.

Variable	Description
$\mathrm{m}^t_{l,p}$	mapping var for logical l and physical p qubits at time step t
mp^t_p	if physical qubit p is mapped to some logical qubit at time step t
$\mathrm{s}^t_{p,p'}$	SWAP variable for physical qubits p, p' at time step t
st^t_p	SWAP-touched variable for physical qubit p at time step t
c^t_i / ac^t_i / dc^t_i	current/advanced/delayed CNOT var for ith CNOT at time step t
$\mathrm{lp}^t_{l,l'}$	logic qubit pair variables for logical qubits l, l' at time step t

In addition to specifying which CNOT gates are chosen in each time step, we also need to specify the CNOT dependencies. We use the DAG generated from the original circuit for computing the CNOT dependencies. We adapt the CNOT dependency constraints by specifying that for every CNOT gate in time step t its predecessors (successors) can be applied at time step t', where $t' \leq t$ ($t' \geq t$). We use two extra CNOT blocks, advanced and delayed CNOTs, which specify if a CNOT gate is mapped in an earlier or later time step. We call this Two-way SAT encoding to emphasize the bidirectional propagation of CNOT dependencies. In the following paragraphs, we describe the main parts of the Algorithm and provide the constraints.

Algorithm 1 Incremental SAT Solving, starting with t=0.

```
 1: for all $l \in [1 \dots \mathrm{nl}]$ do
 2:     ExactlyOne($\mathrm{m}^t_{l,1}, \dots, \mathrm{m}^t_{l,\mathrm{np}}$)
 3: for all $p \in [1 \dots \mathrm{np}]$ do
 4:     AtmostOne($\mathrm{m}^t_{1,p}, \dots, \mathrm{m}^t_{\mathrm{nl},p}$)
 5: MappedPQubits
 6: if t != 0 then
 7:     SwapConstraints and Ancillaries
 8: CNOTConnections and CNOTDependencies
 9: Assumptions
10: Solve instance with assumption $\mathrm{asm}^t$
11: if Instance not satisfied then
12:     repeat from step 1 with t = t + 1
```

Initial Mapping

Let L (P) be a set of logical (physical) qubits in the circuit. Let nl (np) be the number of logical (physical) qubits. In time step 0, we add requirements on the Initial Mapping for logical and physical qubits. Lines 1 to 4 in the Pseudo code add constraints for mapping every logical qubit to a unique physical qubit. We use one-hot encoding for specifying the mapping. We apply ExactlyOne (AtmostOne) constraints for logical (physical) qubit mapping variables. Adding these constraints only at the 0th time step is sufficient for correctness. However, adding these constraints at every time step significantly improved the performance of SAT solvers.

`SwapConstraints`

From time step 1, we use the same mapping variables for handling SWAPs. Adding a SWAP gate changes the mapping between logical and physical qubits. Let CP be the set of all connected physical qubit pairs. The following constraints must ensure that a SWAP gate is only applied on a connected physical qubit pair. The logical qubits mapped on the physical qubit pair must be swapped in the next time step. The qubit mappings for the untouched physical qubits must be propagated. We define two sets of variables to satisfy such constraints. For choosing a SWAP, we define one SWAP variable $\mathrm{s}_{p,p'}$ for each connected physical qubit pair (p, p'). For propagation, we define SWAP-touch variables st_p to specify if the physical qubit p is touched by the SWAP. We specify that 1) Each SWAP variable forces the SWAP-touched physical variables to True; 2) Exactly one of the SWAP variables is set to True; 3) Every SWAP forces exactly two SWAP-touched variables to True. Let S be the set of all SWAP variables. The corresponding boolean constraints are:

$$\bigwedge_{(p,p')\in\mathrm{CP}} (\mathrm{s}^t_{p,p'} \to \mathrm{st}^t_p \wedge \mathrm{st}^t_{p'}) \wedge \mathrm{ExactlyOne}(\mathrm{S}^t) \wedge \mathrm{ExactlyTwo}(\mathrm{st}^t_1, \dots, \mathrm{st}^t_{\mathrm{np}})$$

Based on the chosen SWAP variables, we update the mapping variables. For each SWAP variable $\mathrm{s}_{p,p'}$, we swap the p and p' mapped variables from the previous to the current step.

$$\bigwedge_{(p,p')\in\mathrm{CP}} \bigwedge_{l\in\mathrm{L}} \mathrm{s}^t_{p,p'} \to ((\mathrm{m}^{t-1}_{l,p} \leftrightarrow \mathrm{m}^t_{l,p'}) \wedge (\mathrm{m}^{t-1}_{l,p'} \leftrightarrow \mathrm{m}^t_{l,p}))$$

If a SWAP-touch variable is False, we propagate the corresponding mapping variables.

$$\bigwedge_{p\in\mathrm{P}} \bigwedge_{l\in\mathrm{L}} \neg\, \mathrm{st}^t_p \to (\mathrm{m}^{t-1}_{l,p} \leftrightarrow \mathrm{m}^t_{l,p})$$

`MappedPQubits` and `Ancillaries`

Using additional qubits, traditionally called *ancillaries*, can reduce the total number of SWAPs needed. An *ancillary SWAP* exchanges a mapped qubit with an unmapped qubit. To specify this, we need to keep track of mapped qubits (including at time step 0). We specify that a physical qubit p is mapped to some logical qubit if and only if its mapped variable mp_p is True.

$$\bigwedge_{p \in \mathrm{P}} \mathrm{mp}_p^t \leftrightarrow (\bigvee_{l \in \mathrm{L}} \mathrm{m}_{l,p}^t)$$

We restrict that at least one of the swapped physical qubits is a mapped qubit. With similar constraints, we also provide an option for only non-ancillary SWAPs.

$$\bigwedge_{(p,p') \in \mathrm{CP}} \mathrm{s}_{p,p'}^t \to (\mathrm{mp}_p^t \vee \mathrm{mp}_{p'}^t)$$

`CNOTConnections`

Let CL be the set of all connected logical qubit pairs dervied from the CNOT connections in the input circuit. We require CNOT gates to be applied only on connected physical qubits. Since CNOT gates must be applied to specific logical qubits, we require that the corresponding logical qubits be mapped to connected physical qubits. First, we specify that logical qubit pair variables are true if and only if the physical qubits they are mapped are connected.

$$\bigwedge_{(l,l') \in \mathrm{CL}} \Big(\bigwedge_{(p,p') \in \mathrm{CP}} ((\mathrm{m}_{l,p}^t \wedge \mathrm{m}_{l',p'}^t) \vee (\mathrm{m}_{l,p'}^t \wedge \mathrm{m}_{l',p}^t)) \to \mathrm{lp}_{l,l'}^t \wedge$$
$$\bigwedge_{(p,p') \in \overline{\mathrm{CP}}} ((\mathrm{m}_{l,p}^t \wedge \mathrm{m}_{l',p'}^t) \vee (\mathrm{m}_{l,p'}^t \wedge \mathrm{m}_{l',p}^t)) \to \neg\, \mathrm{lp}_{l,l'}^t \Big)$$

Using logical qubit pair variables, we specify that if a CNOT is mapped then its corresponding logical qubits are connected. We define $\mathcal{D}$ as a dictionary of CNOT indices to logical qubit pairs. Let nc be the number of CNOT gates. The corresponding boolean constraint is:

$$\bigwedge_{\mathrm{nc}}^{i=1} \mathrm{c}_i^t \to \mathrm{lp}_{\mathcal{D}[i]}^t$$

`CNOTDependencies`

For a correct mapping, we need to respect the DAG dependencies of the CNOT gates in a circuit. As discussed earlier, we use advanced and delayed CNOT blocks in each time step to propagate local information globally. Every CNOT is mapped, advanced, or delayed in all time steps, depending on the status of its predecessors (*pre*) and successors (*suc*) in the dependency DAG. If at time step t a CNOT is:

- Mapped: Its predecessors (successors) are either advanced (delayed) or mapped in the same time step.
- Advanced: 1) It is applied or advanced in $t-1$; 2) Its predecessors are also advanced in t.
- Delayed: 1) It is either applied or delayed in $t+1$; 2) Its successors are also delayed in t; 3) In t, either its logical qubits are not connected or one of its predecessors is delayed.

The corresponding boolean constraints are:

$$\bigwedge_{\mathrm{nc}}^{i=1} \big(\,\mathrm{ExactlyOne}(\mathrm{c}_i^t, \mathrm{ac}_i^t, \mathrm{dc}_i^t) \wedge$$
$$\bigwedge_{j \in pre(i)} \mathrm{c}_i^t \to (\mathrm{ac}_j^t \vee \mathrm{c}_j^t) \wedge \bigwedge_{j \in suc(i)} \mathrm{c}_i^t \to (\mathrm{c}_j^t \vee \mathrm{dc}_j^t) \wedge$$
$$\mathrm{ac}_i^t \to (\mathrm{c}_i^{t-1} \vee \mathrm{ac}_i^{t-1}) \wedge \bigwedge_{j \in pre(i)} \mathrm{ac}_i^t \to \mathrm{ac}_j^t \wedge$$
$$\mathrm{dc}_i^{t-1} \to (\mathrm{c}_i^t \vee \mathrm{dc}_i^t) \wedge \bigwedge_{j \in suc(i)} \mathrm{dc}_i^t \to \mathrm{dc}_j^t \wedge \quad \mathrm{dc}_i^t \to (\neg\, \mathrm{lp}_{\mathcal{D}[i]}^t \vee \bigvee_{j \in pre(i)} \mathrm{dc}_i^t)\big)$$

Assumptions for Incremental solving

We specify that CNOT gates cannot be advanced at time step 0 i.e., $\bigwedge_{\mathrm{nc}}^{i=1} \neg\, \mathrm{ac}_i^0$. For using incremental solving in SAT, we use an assumption variable asm^t. At every time step, if the assumption variable is true then CNOT gates cannot be delayed i.e., $\mathrm{asm}^t \leftrightarrow \bigwedge_{\mathrm{nc}}^{i=1} \neg\, \mathrm{dc}_i^t$. For each time step, we call the SAT solver with the assumption variable asm^t as True.

Encoding Size

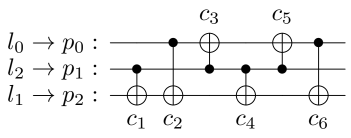
Let l be the number of logical qubits, p be the number of physical qubits, p_e be the number of edges in the physical coupling graph, l_e be the number of edges in the logical graph (from CNOT gates), c be the number of CNOTs, and finally, let k be the make-span. The encoding requires $O(k(lp + p_e + l_e + c))$ variables. Usually, the physical coupling graphs are planar, so the variables required is $O(k(lp + l_e + c))$. Note that, we use the sequential counter encoding for exactly-one constraints, so it can add extra $O(p)$ auxiliary variables. In total, the number of variables is $O(k(lp + l_e + c))$. The encoding requires $O(k(lp + lp_e + l_e p^2 + c))$ clauses. Again, for a planar physical coupling graph this is bounded by $O(k(lp + l_e p^2 + c))$ clauses.

3.1 Additional Functionality

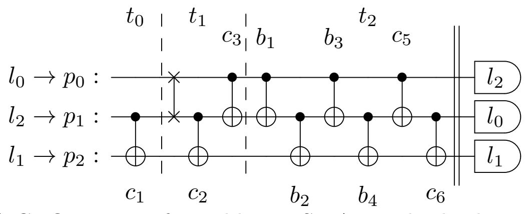
So far, we have not used the semantics of the unary or binary gates (except the SWAP gates). The previous encoding could also be used to map circuits with for instance binary CZ gates instead of CNOT gates. If we take the semantics of the gates into account, there are more opportunities to optimize the circuits. While a complete re-synthesis of the circuit is beyond the scope of this paper, we want to illustrate some known techniques that can further reduce the number of SWAP gates. We emphasize that we now change the optimization problem (by allowing more solutions), and that the extensions are specific for CNOT gates. Our main purpose is to show that our proposed encoding can be easily extended to incorporate these techniques, known as “bridges” and “relaxed dependencies”. We also note that the encoding can be easily restricted to disallow the use of “ancillary qubits”.

Bridges

Using a bridge, one can apply a CNOT on disconnected physical qubits. For instance, the bridge b_1-b_4 in Fig. 5 together implements a CNOT between p_0 and p_2 (which implements c_4 after the preceding SWAP). We limit ourselves to bridges of distance 2. Observe that the bridge introduces 3 extra CNOT gates, so the cost is the same as a SWAP gate. However, the result is different, since a bridge does not swap the qubits. This might be an advantage, depending on the rest of the circuit. In [11], it was shown that using bridges can reduce the overall CNOT count.



(a) CNOT gates after initial mapping.

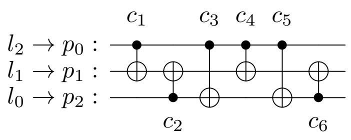


(b) CNOT gates after adding a SWAP and a bridge gate.

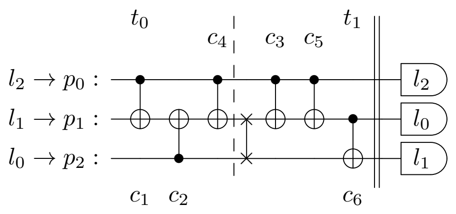
Figure 5 Mapping reduced circuit using a SWAP and a bridge gate.

We adapted our optimal Two-Way SAT encoding, by allowing to add either a single bridge or a single SWAP gate at each time step. If a bridge was added, the corresponding CNOT gate is regarded as scheduled. So both options cost exactly 3 CNOT gates. The SAT solver will find a solution with the minimal sum of bridge or SWAP gates. Our experiments will show that we indeed find better solutions with bridges.

Relaxed Dependencies



(a) CNOT gates after initial mapping.



(b) CNOT gates after adding a SWAP with commutation.

Figure 6 Mapping reduced circuit using a SWAP and relaxed dependencies.

The authors of [12] consider gate commutation rules for quantum layout mapping. Commutation and cancellation rules on R_Z and CNOT gates are also used in [20], to reduce the number of H-gates. For instance, two subsequent CNOT gates on the same control qubit, or on the same data qubit, can be commuted without changing the semantics. Also, single Z-like gates (like the Z-, S-, T- and R_Z-gates) commute with the control bit of a CNOT, while X-like gates (like the X- and R_X-gates) commute with its data bit.

This added freedom can be exploited: by permuting the CNOT gates, they can be grouped in a convenient manner, so less SWAP gates are needed. For instance, in Fig. 6a, c_3 and c_4 can be commuted, since they share their control bit. As a result, we can now find a solution that requires only 1 SWAP gate (Fig. 6b), while still respecting the linear coupling graph.

The gate commutation rules can be incorporated in our optimal Two-way SAT encoding by computing a "relaxed" dependency graph. In the example above, we consider c_2 as a dependency for c_3 and c_4, but c_3 and c_4 are considered independent. We stress that the relaxed dependencies must also take the unary gates into account (for instance c_2 and c_4 cannot be commuted, since there is a T-gate in between, cf. Fig. 1).

In our tool, we compute the relaxed dependency graph before removing the unary gates. We then generate the encoding as presented before, replacing the dependency graph with the relaxed dependency graph. This guarantees an optimal Layout Mapping (minimal number of SWAPS) given the specified commutation rules. Our experiments show that relaxed dependencies can indeed provide better solutions.

Non-Ancillary Mapping

The actual cost of ancillary qubits in practical quantum computing depends on the context. We provide optimal layout synthesis without any ancillary SWAPs as an option. Note that the resulting encoding may require more SWAP gates than when allowing ancillary qubits. This option can be encoded in our Two-Way SAT encoding, by simply restricting the SWAP gates to cases where both the physical qubits are already mapped.

3.2 Design Choices

Redundant Cardinality Constraints for Mapping Variables

Specifying Exactly-One constraints (EO) on logical qubits and At-Most-One constraints (AO) on physical qubits in the initial time step would be sufficient for correctness. Note that once the mapping variables are set in the initial time step, the information on bijectivity is propagated to next time steps, based on the chosen SWAP variables. However, observe that unrelated to the choice of SWAP variables, some invariants apply to mapping variables in all time steps. Essentially, the EO and AO constraints are orthogonal to the SWAP variable assignment. We observed that adding such redundant constraints at each time step significantly improved solving times. Apparently, this local information can be exploited by the SAT solver during clause learning or unit propagation. Since we added mostly binary clauses, we conjecture that the improved solving time is due to improved unit propagation.

Two-Way Encoding vs Explicit CNOT Constraints

In this paper, we encoded CNOT constraints using a Two-Way encoding instead of explicit CNOT constraints. We chose Two-Way encoding for two main reasons. First, we encode transitive closure for predecessors/successors of CNOT gates. Explicit constraints for CNOT dependencies result in two challenges:

- Specifying that the predecessors (successors) of a CNOT gate can not be scheduled in later (earlier) time steps results in a quadratic blow-up in the CNOT gates.
- Specifying that the predecessors (successors) of a CNOT gate must be scheduled in earlier (later) time steps results in long clauses.

On the other hand, the Two-Way encoding expresses this bidirectional propagation implicitly using clauses linear in the number of CNOT gates. Second, in incremental solving, in each iteration, we need to specify that the goal is reached in the final time step. Using the Two-Way encoding, we can simply specify that no CNOTs are delayed in the final step. If we encoded the CNOT constraints explicitly, we would need to specify that every CNOT is scheduled at some time step. To avoid large clauses, one would need to use auxiliary variables similar to advanced/delayed variables to keep track of the scheduled CNOTs across time steps. In our tool, we provide the encoding with explicit CNOT constraints as an option.

4 Experimental Evaluation

4.1 Experiment Design

We have extended our tool Q-Synth v1 (Quantum Synthesizer) to include the Two-Way Parallel SAT encoding. We provide an open-source tool Q-Synth2[4] that implements the SAT encoding and the additional options. For any option chosen, our tool synthesizes

[4] Q-Synth v2 tool with source code, benchmarks, and scripts `https://github.com/irfansha/Q-Synth`

a mapped circuit with the (provably) optimal number of additional SWAP+bridge gates. We use pysat [9] for generating and solving SAT instances incrementally. For cardinality constraints, we use the sequential counter from pysat. As a backend for our experiments, we use Cadical-1.53 [2], a state-of-the-art SAT solver. One can easily experiment with other SAT solvers in our tool using the pysat interface. When optimizing the SWAP count, our tool refutes all $k-1$ SWAP+bridge additions if k is optimal. We report a timeout if an optimal solution is not found within the time limit. We check equivalence between the original circuits and our mapped circuits with QCEC[5] [5] for correctness.

We design 3 experiments. Our goal is to investigate the effectiveness of our SAT encoding compared to the current leading tools. We also compare various additional techniques discussed in 3.1. For comparison, we consider state-of-the-art tools TB-OLSQ2 (near-optimal) [16] and Qiskit's SABRE (heuristic). For TB-OLSQ2, we enable the best options i.e., SWAP optimization and upper bound computation by SABRE, with z3 (v4.12.1.0) [8] as the backend. TB-OLSQ2 can provide intermediate non-optimal results. We only report the final (near-optimal) solution when it terminates. If the tool does not terminate within the time limit, we report it as a timeout. For SABRE, we use the first 1000 seeds for the SABRE layout and take the minimum SWAPs generated by any seed. We also compare our results with other leading tools in Section 1.

Experiment 1: Standard Benchmarks on Large Platforms

We consider the standard benchmarks from papers [16, 26, 28] with 23 instances in total. The benchmark set contains circuits of up to 54 qubits and 270 CNOT gates. The circuits are mapped to the current NISQ processors, Sycamore with 54 qubits [1], Rigetti with 80 qubits[6] and Eagle with 127 qubits [6]. We compare with the tools TB-OLSQ2 and SABRE, with a time limit of 12000 seconds (3hr 20 minutes) for each instance and an 8 GB memory limit.

Experiment 2: Deep VQE Benchmarks on Large Platforms

From our experiments and also consistent with the literature [16], most of the benchmarks from Experiment 1 need at most 10 SWAPs on standard platforms. To investigate the performance on deep circuits that need many SWAPs, we use a set of 10 random circuits composed using operators from the Variational Quantum Eigensolver (VQE) algorithm presented in [17]. Our second benchmark set consists of 10 (8 qubit) circuits with up to 79 CNOT gates. Due to many interactions between the qubits, the number of SWAPs needed to map onto the standard quantum platforms is high. We use the same time and memory limits as in Experiment 1. In both Experiments 1 and 2, we denote a timeout with TO. Here we focus on comparison with TB-OLSQ2 and report both SWAP count and circuit depth.

Experiment 3: Effectiveness of Additional Functionality

In this experiment, we compare 4 combinations of SWAPs (S), bridges (B), and relaxed dependencies (R): 1) S 2) S+B 3) S+R 4) S+B+R. From our two benchmark sets, we consider all the circuits with 14 or fewer qubits and map them onto the standard Melbourne platform of 14 qubits. We give a time limit of 600 seconds (or 10 minutes) and an 8 GB of memory.

[5] Munich Quantum Toolkit QCEC `https://github.com/cda-tum/mqt-qcec`
[6] Rigetti Computing `https://www.rigetti.com`

Of the 24 instances generated, we drop qft_8 which times out in all 4 combinations. For the rest of the 23 instances, we report SWAPs+bridges for each combination. Note that every additional SWAP or bridge adds exactly 3 extra CNOTs to the mapped circuit.

4.2 Results

Table 2 Experiment 1: Number of SWAPs required for mapping circuits with QS2: Q-Synth2 (SWAP-optimal), TO2: TB-OLSQ2 (near optimal), and SB: SABRE (heuristic) tools on different platforms. Syc: Sycamore (54), Rig: Rigetti (80), Eagle (127) and label or(3/6) represents a circuit "or" with 3 qubits and 6 CNOT gates.

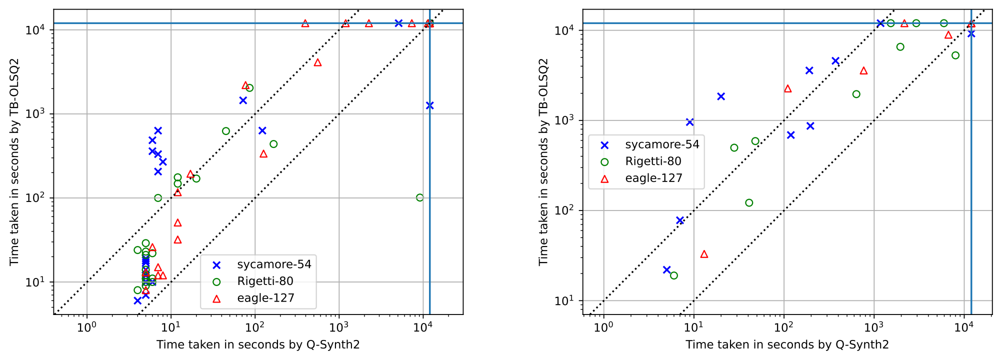
platform (qubits):	Sycamore (54)			Rigetti (80)			Eagle (127)		
Circuit(q/cx) / Tool	**QS2**	TO2	SB	**QS2**	TO2	SB	**QS2**	TO2	SB
or(3/6)	2	2	3	2	2	2	2	2	2
adder(4/10)	0	0	0	0	0	0	2	2	2
qaoa5(5/8)	0	0	1	0	0	0	0	0	1
4mod5-v1_22(5/11)	3	3	4	3	3	5	3	3	6
mod5mils_65(5/16)	6	6	7	6	6	7	6	6	8
4gt13_92(5/30)	10	10	15	10	10	15	**13**	TO	15
tof_4(7/22)	1	1	3	1	1	11	3	3	5
barenco_tof4(7/34)	5	5	18	6	6	17	8	8	17
qft_8(8/56)	**9**	TO	15	TO	TO	12	TO	TO	23
tof_5(9/30)	1	1	3	1	1	5	3	3	12
mod_mult55(9/40)	6	6	9	**7**	8	16	**12**	TO	20
barenco_tof5(9/50)	6	6	10	8	8	19	**12**	TO	20
vbe_adder3(10/50)	7	7	8	8	8	14	10	10	33
rc_adder6(14/71)	TO	**8**	16	8	8	35	TO	TO	51
ising_model10(16/90)	0	0	0	0	0	0	0	0	0
queko(16/15)	0	0	1	0	0	2	0	0	0
queko(16/29)	0	0	5	**0**	1	12	2	2	14
queko(16/44)	0	0	7	**0**	1	25	2	2	37
queko(16/58)	0	0	12	**0**	1	20	**4**	TO	41
queko(16/87)	**0**	1	10	**0**	1	30	**4**	TO	36
queko(16/101)	0	0	18	**0**	1	43	TO	TO	36
queko(54/54)	**0**	1	12	1	1	31	TO	TO	47
queko(54/270)	**0**	1	183	TO	TO	302	TO	TO	428
Total solved of 23	22	22	23	21	21	23	**18**	13	23

Experiment 1

Table 2 reports the number of SWAPs added. Both on Sycamore and Rigetti, TB-OLSQ2 mostly reports optimal SWAP count (while not proving optimality). There are 10 instances where it reports near-optimal solutions i.e., only 1 extra SWAP gate or times out. The difference is more significant on the larger 127-qubit Eagle platform. Q-Synth2 solved 5 more instances optimally where TB-OLSQ2 times out. Figure 7a provides the scatter plot of the time taken by Q-Synth2 and TB-OLSQ2. Except for two instances with rcadder6 (14 qubits) on Sycamore and Rigetti, we significantly outperform TB-OLSQ2 on all three platforms. In several instances, Q-Synth2 is one or two orders of magnitude faster while proving optimality. In the case of two instances with rcadder6, being a 14 qubit circuit, our tool takes time to refute the k-1 number of optimal SWAPs. While the heuristic tool SABRE always returns a mapping within 2 minutes, it also adds too many additional SWAPs. This observation is consistent with the literature [16].

Table 3 Experiment 2: Additional SWAPs (s.) and Depth (d.) of mapped VQE circuits on different platforms with QS2: Q-Synth2 (SWAP-optimal) and TO2: TB-OLSQ2 (near optimal).

platform	Syc (54)				Rig (80)				Eagle (127)			
	QS2		TO2		**QS2**		TO2		**QS2**		TO2	
Circuit(q/cx)	s.	d.	s.	d.	s.	d.	s.	d.	s.	d.	s.	d.
vqe(8/18)	2	34	2	**33**	2	36	2	**33**	3	38	3	**35**
vqe(8/39)	**4**	65	5	**62**	**6**	68	7	**63**	**7**	65	9	**64**
vqe(8/40)	**6**	70	7	**67**	7	**67**	7	69	**10**	76	12	**68**
vqe(8/47)	**8**	85	10	**83**	10	84	10	**83**	**14**	92	17	**86**
vqe(8/48)	6	90	6	**84**	8	94	8	**89**	**12**	94	TO	TO
vqe(8/52)	**9**	90	11	**87**	**11**	90	TO	TO	TO	TO	TO	TO
vqe(8/63)	**10**	101	11	**99**	13	102	13	102	TO	TO	TO	TO
vqe(8/71)	TO	TO	**16**	**111**	**15**	**112**	18	113	TO	TO	TO	TO
vqe(8/78)	**14**	**129**	TO	TO	**17**	**136**	TO	TO	TO	TO	TO	TO
vqe(8/79)	11	149	11	**146**	**16**	**151**	TO	TO	TO	TO	TO	TO
Solved (10)	9		9		**10**		7		**5**		4	



(a) Experiment 1. **(b)** Experiment 2.

Figure 7 Scatter plots of time taken by TB-OLSQ2 and Q-Synth2.

Experiment 2

Table 3 reports the number of SWAPs added and circuit depth for the mapped VQE circuits. On all platforms, Q-Synth2 solves 15 more instances SWAP-optimally out of the 24 instances solved by either of the tools. TB-OLSQ2 in general reports better circuit depth compared to Q-Synth2. Interestingly in two instances, vqe(8/40) and vqe(8/71), Q-Synth2 reports better circuit depth. This shows that TB-OLSQ2 is near optimal in both SWAP additions and circuit depth. Overall Q-Synth2 also reports near-optimal depth while optimizing additional SWAPs. Figure 7b shows the scatter plot of time for Experiment 2. Except for two instances with vqe(8/71) on Sycamore and Rigetti, Q-Synth2 significantly outperforms TB-OLSQ2.

Table 4 Experiment 3: Number of SWAPs+bridges required for mapping deep VQE circuits on Melbourne platform (14-qubits) in 600 seconds with Q-Synth2 with combinations of S: Swaps, B: bridges, and R: relaxed dependencies.

Circuit(q/cx)	S	SB	SR	SBR
or(3/6)	2	2	**1**	**1**
adder(4/10)	0	0	0	0
qaoa5(5/8)	0	0	0	0
4mod5_22(5/11)	3	**2**	**2**	**2**
mod5mils65(5/16)	6	**4**	**4**	**4**
4gt13_92(5/30)	10	**8**	**8**	**8**
tof_4(7/22)	1	1	1	1
barencof4(7/34)	5	5	5	5
tof_5(9/30)	1	1	1	1
modmult55(9/40)	7	7	7	7
barencof5(9/50)	6	6	6	6
vbe_adder(10/50)	8	8	**6**	**6**
rcadder6(14/71)	9	**8**	9	**8**
vqe(8/18)	2	2	2	2
vqe(8/39)	6	6	6	6
vqe(8/40)	7	7	7	**6**
vqe(8/47)	8	8	8	8
vqe(8/48)	7	**6**	7	**6**
vqe(8/52)	10	10	10	10
vqe(8/63)	12	12	12	12
vqe(8/71)	13	**12**	13	**12**
vqe(8/78)	17	15	16	**14**
vqe(8/79)	15	**13**	14	**13**

Experiment 3

Table 4 reports the number of SWAPs+bridges on the Melbourne platform, with a time out of 10 minutes. Both bridges and relaxed dependencies can reduce the optimal SWAP+bridge additions. We observe that both techniques together can reduce CNOT count further. For instance, vqe(8/78) only needs 14 SWAPs+bridges i.e., 9 fewer CNOTs compared to only adding SWAPs. If we drop any of the options, the optimal CNOT count is higher.

4.3 Discussion

Comparison to OLSQ, OLSQ2 and TB-OLSQ2

In [16], authors showed that TB-OLSQ2 significantly outperforms both OLSQ and OLSQ2. Because of the lack of grouping in OLSQ and OLSQ2, the make-span of SMT instances generated is very large. Since our tool outperforms TB-OLSQ2, we do not report results from the other two directly. TB-OLSQ2 optimization routines can also be used in our tool to avoid hard unsatisfiable instances when optimality is not needed.

Comparison to Q-Synth v1 with Classical Planning

In [26], we showed that Q-Synth v1 based on Classical Planning outperformed both OLSQ and QMAP. While the approach scales well for mapping circuits up to 9 qubits onto 14 qubits platforms, larger circuits are out of reach. For instance, Q-Synth v1 timed out on rcadder6 (Table 4) with 3 hours. Q-Synth2 maps the same instance optimally within 5 minutes onto the 14-qubit Melbourne platform. Q-Synth v1 does not scale well to the other larger quantum platforms. Mapping individual CNOTs in Q-Synth v1 results in long plan length. As discussed in the same paper, long plan lengths increase the difficulty of planning.

Comparison to QMAP and SATMAP

QMAP [33, 31] employs an SMT encoding that grows exponentially with the number of physical qubits. Even using subarchitectures, QMAP is unable to map circuits greater than 7 qubits. SATMAP [19] on the other hand, encodes the Layout Synthesis as a MAXSAT problem to minimize the number of SWAPs. It allows the addition of one SWAP before every CNOT and uses MAXSAT solvers to minimize the number of SWAPS. As shown in [16], it produces suboptimal solutions and runs out of time even for moderate circuits.

Comparison with Dynamic Programming Approach

In [11], the authors provided an exact and a heuristic approach for adding SWAPs and bridge gates. With commutation rules, they showed that using bridges can further reduce the optimal CNOT additions. Our experiments are consistent with the authors' observations. Their exact approach already takes 12 minutes to map a 6-qubit circuit to 6-qubit platforms and grows exponentially with the number of qubits.

Comparison to SABRE

As observed in Experiment 1, it is clear that heuristic approaches such as SABRE add too many SWAPs. Adding many SWAPs not only increases the 2-qubit gate count but also increases circuit depth. However, heuristic approaches have their place in the quantum compilation pipeline. As the number of qubits on quantum processors increases, it is necessary to employ a hybrid strategy with heuristic and exact approaches. For instance, one could use SABRE to quickly find a reasonable initial mapping. Given such a mapping, one can synthesize a SWAP-optimal mapped circuit using Q-Synth2. As in TB-OLSQ2, we could use heuristic approaches to get quick upper bounds for near-optimal solving.

5 Conclusion

In this paper, we showed that parallel plans can be adapted to preserve SWAP optimality in layout synthesis. We have encoded the parallel planning problem directly in SAT. We propose a Two-Way encoding, in which information is propagated both forward and backward, for efficiency. We also demonstrated that our Two-Way SAT encoding is compatible with other techniques, like bridges and gate commutation rules.

The technique is implemented in the open-source tool Q-Synth2 for scalable and optimal layout synthesis of deep circuits. We can optimally map 8-qubit circuits that require up to 14 SWAPs onto a 127-qubit platform. We significantly outperform leading near-optimal tools while still guaranteeing that the resulting mapping is optimal.

References

1 Frank Arute et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. doi:10.1038/s41586-019-1666-5.

2 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53. University of Helsinki, 2020. URL: https://api.semanticscholar.org/CorpusID:220727106.

3 Kyle E. C. Booth. Constraint programming models for depth-optimal qubit assignment and swap-based routing (short paper). In *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 43:1–43:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CP.2023.43.

4 Sebastian Brandhofer, Ilia Polian, and Kevin Krsulich. Optimal qubit reuse for near-term quantum computers. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023*, pages 859–869. IEEE, 2023. doi:10.1109/QCE57702.2023.00100.

5 Lukas Burgholzer and Robert Wille. Advanced equivalence checking for quantum circuits. *IEEE TCAD*, 40(9):1810–1824, 2021. doi:10.1109/tcad.2020.3032630.

6 Jerry Chow, Oliver Dial, and Jay Gambetta. Ibm quantum breaks the 100-qubit processor barrier. *IBM Research Blog*, 2, 2021. URL: https://www.ibm.com/quantum/blog/127-qubit-quantum-processor-eagle.

7 Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Lev Bishop, Colm A. Ryan, Steven Heidel, Niel de Beaudrap, John Smolin, Jay M. Gambetta, and Blake R. Johnson. Open quantum assembly language. *ACM Transactions on Quantum Computing Journal*, 2022. URL: https://www.amazon.science/publications/open-quantum-assembly-language.

8 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *TACAS Proceedings*, LNCS 4963, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

9 Alexey gnatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018. doi:10.1007/978-3-319-94144-8_26.

10 Stefan Hillmich, Alwin Zulehner, and Robert Wille. Exploiting quantum teleportation in quantum circuit mapping. In *ASPDAC '21*, pages 792–797. ACM, 2021. doi:10.1145/3394885.3431604.

11 Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration*, 70:43–50, 2020. doi:10.1016/j.vlsi.2019.10.004.

12 Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. Quantum circuit compilers using gate commutation rules. In *ASPDAC*, pages 191–196. ACM, 2019. doi:10.1145/3287624.3287701.

13 Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of KR-96*, pages 374–384, November 1996. URL: https://henrykautz.com/papers/plankr96.pdf.

14 Henry A Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992. URL: http://www.cs.cornell.edu/selman/papers/pdf/92.ecai.satplan.pdf.

15 Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for NISQ-era quantum devices. In *ASPLOS*, pages 1001–1014. ACM, 2019. doi:10.1145/3297858.3304023.

16 Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. Scalable optimal layout synthesis for NISQ quantum processors. In *DAC*, 2023. doi:10.1109/DAC56929.2023.10247760.

17 Marco Majland, Patrick Ettenhuber, and Nikolaj Thomas Zinner. Fermionic adaptive sampling theory for variational quantum eigensolvers. *Phys. Rev. A*, 108:052422, November 2023. doi:10.1103/PhysRevA.108.052422.

18 Joao Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability*, 336:133–182, 2021. doi:10.3233/FAIA200987.

19 Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Qubit mapping and routing via MaxSAT. In *MICRO*, pages 1078–1091. IEEE, 2022. doi:10.1109/MICRO56248.2022.00077.

20 Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1), May 2018. doi:10.1038/s41534-018-0072-4.

21 Michael A. Nielsen and Isaac L. Chuang. *Quantum circuits*, pages 171–215. Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.008.

22 Tom Peham, Lukas Burgholzer, and Robert Wille. On optimal subarchitectures for quantum circuit mapping. *ACM Trans. on Quant. Computing*, 2023. doi:10.1145/3593594.

23 Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006. doi:10.1016/J.ARTINT.2006.08.002.

24 Irfansha Shaik and Jaco van de Pol. Q-Synth. Software, version 2.0., IFD (Innovation Fund Denmark), swhId: swh:1:dir:be31c57364bd541c6b65afac80603e9004cf4008 (visited on 2024-07-31). URL: https://github.com/irfansha/Q-Synth.

25 Irfansha Shaik and Jaco van de Pol. Q-Synth v2.0 release. Software, version 2.0., IFD (Innovation Fund Denmark), swhId: swh:1:dir:be31c57364bd541c6b65afac80603e9004cf4008 (visited on 2024-07-31). URL: https://github.com/irfansha/Q-Synth/releases/tag/Q-Synth-v2.0-SAT2024.

26 Irfansha Shaik and Jaco van de Pol. Optimal layout synthesis for quantum circuits as classical planning. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD 2023, San Francisco, CA, USA, October 28 - Nov. 2, 2023*, pages 1–9. IEEE, 2023. doi:10.1109/ICCAD57390.2023.10323924.

27 Amisha Srivastava, Chao Lu, Navnil Choudhury, Ayush Arunachalam, and Kanad Basu. Search space reduction for efficient quantum compilation. In *Proceedings of GLSVLSI-23*, pages 109–114. ACM, 2023. doi:10.1145/3583781.3590223.

28 Bochen Tan and Jason Cong. Optimal layout synthesis for quantum computing. In *IEEE/ACM ICCAD*, pages 137:1–137:9. IEEE, 2020. doi:10.1145/3400302.3415620.

29 Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *IEEE Trans. Computers*, 70(9):1363–1373, 2021. doi:10.1109/TC.2020.3009140.

30 Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Temporal planning for compilation of quantum approximate optimization circuits. In *Proceedings of IJCAI-17*, pages 4440–4446, 2017. doi:10.24963/ijcai.2017/620.

31 Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *DAC-19*, page 142. ACM, 2019. doi:10.1145/3316781.3317859.

32 Alwin Zulehner, Hartwig Bauer, and Robert Wille. Evaluating the flexibility of a* for mapping quantum circuits. In *Reversible Computation - 11th International Conference, RC 2019, Lausanne, Switzerland, June 24-25, 2019, Proceedings*, volume 11497 of *Lecture Notes in Computer Science*, pages 171–190. Springer, 2019. doi:10.1007/978-3-030-21500-2_11.

33 Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE TCAD ICS*, 38(7):1226–1236, 2019. doi:10.1109/TCAD.2018.2846658.

Revisiting SATZilla Features in 2024

Hadar Shavit
Chair for AI Methodology, RWTH Aachen University, Germany

Holger H. Hoos
Chair for AI Methodology, RWTH Aachen University, Germany
Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands
Department of Computer Science, University of British Columbia, Vancouver, Canada

Abstract

Boolean satisfiability (SAT) is an $\mathcal{NP}$-complete problem with important applications, notably in hardware and software verification. Characterising a SAT instance by a set of features has shown great potential for various tasks, ranging from algorithm selection to benchmark generation. In this work, we revisit the widely used SATZilla features and introduce a new version of the tool used to compute them. In particular, we utilise a new preprocessor and SAT solvers, adjust the code to accommodate larger formulas, and determine better settings of the feature extraction time limits. We evaluate the extracted features on three downstream tasks: satisfiability prediction, running time prediction, and algorithm selection. We observe that our new tool is able to extract features from a broader range of instances than before. We show that the new version of the feature extractor produces features that achieve up to 26% lower RMSE for running time prediction, up to 3% higher accuracy for satisfiability prediction, and up to 15 times higher closed gap for algorithm selection on benchmarks from recent SAT competitions.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Satisfiability, feature extraction, running time prediction, satisfiability prediction

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.27

Supplementary Material *Software (Source Code)*: `https://github.com/hadarshavit/revisiting_satzilla` [16], archived at `swh:1:dir:a06a5e5ad5473cd07b6a423a1dc5491b54af7c61`

Funding This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation program under GA No. 952215. HH gratefully acknowledges support through an Alexander-von-Humboldt Professorship in Artificial Intelligence.

Acknowledgements We would like to thank Matthias König and Anja Jankovic for the discussions. We would also like to thank the reviewers who gave a valuable feedback.

1 Introduction

The Boolean satisfiability problem (SAT) is an important problem in computer science from both theoretical and practical viewpoints. Common usages of SAT include hardware and software verification, cryptography, and more. However, as SAT is $\mathcal{NP}$-complete, it takes substantial time and computing power to solve it, which becomes prohibitively expensive as formulas become larger. In order to deepen our understanding of SAT itself and develop better SAT solvers, it is of crucial importance to be able to describe SAT instances via an informative set of features.

Some of the most widely adopted such features are the SATZilla features [13, 7]. These are a fixed set of features that are calculated from the DIMACS CNF representation of a SAT instance. The SATZilla features consist of multiple groups, ranging from basic syntactic features describing the formula, such as its number of variables and clauses, to more complicated features, such as probing features derived from short runs of SAT solvers. Another type of features are based on statistics of graph representations of a given formula.

The SATZilla features have been successfully used in various domains, such as empirical performance models (EPMs; also known as performance or running time prediction) and algorithm selection [17, 18], algorithm configuration [6], and benchmark generation [8]. The features are also used for caching in CDCL-based model counting solvers [14] and in a variety of SAT solvers that incorporate machine learning techniques [4] However, the SATZilla features in their latest version date back to 2012. Since then, the SAT community has undergone various changes. Most notably, SAT instances that we typically encounter today have a larger number of variables and clauses, thus taking significantly more time and memory to preprocess. Currently, the existing SATZilla feature extraction tool is unable to compute many of these features because of time and memory limitations.

In this paper, we revisit the SATZilla features and introduce a new version of the feature extraction tool. First, we replace the underlying solvers and the preprocessor with their most up-to-date versions. We then fix compilation errors and other memory errors related to dealing with larger formulas. Finally, we allow the user to set the time limits for feature computation. We compare the performance of our new tool with the old one in two SAT competitions. We measure the running times and the number of extracted features to check for performance gains of our new tool. We then evaluate the extracted features on three downstream tasks: satisfiability prediction, performance prediction and algorithm selection. We show that our new tool yields an important advantage in performance compared to the old tool across all three tasks.

The rest of the paper is organised as follows. We give a historical overview of the development and applications of the SATZilla features in Section 2. We then introduce the technical definitions, as well as the standard methodological pipeline in Section 3, wrapping it up with the contributions of this study. The results are presented in Section 4, and conclusions drawn in Section 5.

2 Related work

The SATZilla features were first introduced by Nudelman et al. [13] to construct EPMs, *i.e.*, machine learning models that predict the running time of various SAT solvers given the features representing SAT instances. The authors identified key features that contribute the most towards having a good EPM prediction. Consequently, they used the EPMs as a basis for algorithm selection, in which the algorithm that is selected corresponds to the one with the lowest predicted running time. They leveraged the *performance complementarity* phenomenon of SAT solvers, where no SAT solver dominates all others over all instances. Therefore, selecting the best solver for each instance results in substantially better performance compared to choosing any standalone solver for all instances. The SATZilla features were also successfully used for the satisfiability prediction task for SAT instances from various distributions [2].

Further developments in algorithm selection led to the 2007 version of the SATZilla algorithm selector [17], which combined running time and satisfiability prediction with other improvements. It won multiple medals in the SAT competition. This shows that extracting features from SAT instances can also speed up the solving process, and not only help understanding SAT. A newer version of the SATZilla algorithm selector was introduced in 2012 [18], winning multiple awards as well. It used a random forest as a predictor, and introduced an ensemble of pairwise classifiers to establish a ranking of the solvers, instead of predicting the running times directly. Additional features were introduced thereafter, revealing further important information about SAT instances.

Another common usage of the SATZilla features is (model-based) algorithm configuration. To this end, the hyperparameters of SAT solvers are optimised such that their performance is as good as possible on all instances. As running SAT solvers is computationally expensive, a surrogate model is employed; it takes as an input a configuration of hyperparameters and instance features and predicts the performance of the SAT solver using a given configuration on a given instance. The instance features boost the accuracy of the surrogate model. An example of that is SMAC [6], which successfully used the SATZilla features to optimise the performance of a wide range of SAT solvers on various benchmarks.

Last but not least, feature-based EPMs are proven to be useful for benchmark generation [8]. In this context, given a new instance, we compute the features, which is usually cheaper than running a SAT solver, and use the EPM to predict whether the instance is hard (or not) for the solvers at hand.

3 SATZilla features

The SATZilla features describe the SAT formula using various representations and statistics. We briefly introduce three graph representations of a SAT formula, as undirected graphs are a meaningful representation of SAT, maintaining the permutation invariance: a) variable graph: nodes are variables, an edge exists if variables appear in the same clause; b) clause graph: nodes are clauses, an edge exists when two clauses share a negated literal; and c) variable-clause graph: nodes are variables and clauses, an edge exists between a variable node and a clause node if the variable appears in the clause.

Feature computation starts with the *preprocessing* of the formula. This step, performed before solving an instance, renders the formula more accessible for SAT solvers. This means that the features are also computed on the version of the formula that is close to the one seen by the solvers. We believe that all these aspects can be boosted by using a modern preprocessor. Features are classically computed using the SATELITE preprocessor. We instead use the SBVA preprocessor, suggested by the winning solver from the 2023 SAT Competition. SBVA is also able to terminate after a set cutoff time, allowing for partial preprocessing, while SATELITE does not include this functionality. The preprocessed formula can then be directly used by a SAT solver without additional preprocessing within the solver, which improves the performance of algorithm selection.

Following the preprocessing, the *feature extraction* begins. There are ten feature groups that can be extracted. We note that we describe the feature groups according to their implementation in the SATZilla feature extraction tool, not according to their definitions from the corresponding paper [7]. We point out that all feature groups include the time required to compute the features in the group.

Preliminary features include the number of variables and clauses before/after preprocessing. The running time of this feature group includes the preprocessing time and the time required to read the formula. This group contains 7 features.

Basic features are cheap features that provide a basic description of the formula. They consist of the variable-clause ratio, the ratio between positive and negative literals in each clause, the number of unary, binary and ternary clauses, as well as statistics on clause nodes in the variable-clause graph. This group contains 15 features.

KLB are expensive features that include the node degree statistics of the variable nodes in the variable-clause graph, and the ratio of positive to negative occurrences of each variable. They also include measures for the proximity to Horn formula, such as the fraction of Horn clauses and statistics on the number of times each variable appears in a Horn clause. This group contains 21 features.

Clause graph (CG) features are expensive features that contain statistics on the degree of the nodes in the clause graph, as well as the clustering coefficient. This group contains 11 features.

Diameter features contain information on the diameter of the variable graph, which is the shortest path between each pair of nodes in the graph. This group contains 6 features.

DPLL probing (or unit propagation) features are computed by running the DPLL algorithm for various depths and measuring the number of unit propagations at each depth. This group contains 6 features.

Lobjois features are an estimation of the size of the search space. They are computed by running the DPLL algorithm multiple times until a contradiction is found. Then, the average depth of the contradictions is the log-estimation of the search space. This group contains 3 features, which are all based on the work of Lobjois and Lemaître [11].

Survey propagation features are based on computing statistics on the following probabilities returned by the VARSAT [5] solver: a probability of each variable to be assigned to True, to False, and to be unconstrained. This group contains 19 features.

Clause learning (CL) features are based on running a CDCL solver (ZChaff rand [12] in the 2012 version, CadiCal [1] in our new version) for two seconds. We measure the number and length of learned clauses for every 1000 decisions, and compute the statistics of those values. This group contains 19 features.

Local search (LS) features are obtained by running two local search solvers many times, each time up to 10000 steps, and computing statistics on those runs. In the 2012 version, the local search solvers are GSAT and SAPS. We instead use GSAT and Sparrow 2011 in our new version. This group contains 24 features.

Linear programming (LP) features are based on solving a relaxed version of the SAT formula, where $C_1 \wedge C_2 \wedge \cdots \wedge C_N$ is a Boolean formula with N CNF clauses $C_1 \ldots C_N$ over Boolean variables x_j. We now consider linear programming variables x_j and solve the following linear programming problem: Maximise $\sum_{i=1}^{N} \sum_{l \in C_i} v(l)$, where the value $v(l)$ of literal l is defined as $v(x_j) = x_j$, $v(\neg x_j) = 1 - x_j$, while keeping $\sum_{l \in C_i} v(l) \geq 0$ for each C_i and $0 \leq x_j \leq 1$ for each x_j. This means that every variable has a value between 0 and 1, each clause has a value which is the sum of values of all literals, and the value of the formula is the sum of values of all clauses. The goal is to maximise the value of the formula, while keeping the value of every clause positive. Finally, statistics on the linear programming solution are extracted. In the new version, we upgrade the linear programming solver package, lp solve. This group contains 7 features.

We note that the basic, KLB and clause graph features are computed sequentially, with each feature group being dependent on the successful computation of the previous groups; *e.g.*, if the computation of the basic features fails, the KLB and clause graph features will not be computed at all.

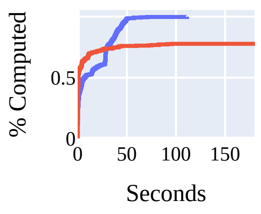
Another change we applied to the feature extraction tool is allowing a more precise timeout setting. In the 2012 version, the time limits were hard-coded and set to high values (for example, 1200 seconds for preprocessing). This can cause many feature computations to simply terminate without computing any features at all. To this end, we adjust the code in the new version to allow the user to set the time limits through a command line argument.

4 Experiments

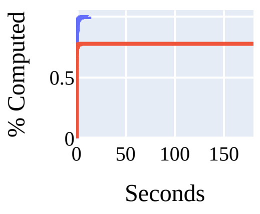
We evaluate our new SATZilla feature extraction tool on the formulas stemming from two latest (2022 and 2023) SAT Competitions. We first look at the feature extraction times and then evaluate the features on three downstream tasks: satisfiability prediction, running time prediction, and algorithm selection. To assess the advantage of using the new version of the extraction tool, we extract the features using both our new version and the 2012 version of the tool, with a time limit of 180 seconds per feature group.

We use a cluster of 18 nodes, each equipped with 2 AMD EPYC 7543 32-core CPUs with 256 MB L3 cache. Each node also has 1TB of memory. The cluster is running on a Rocky Linux 9.4 operating system. We measure running times using the runsolver tool [15].

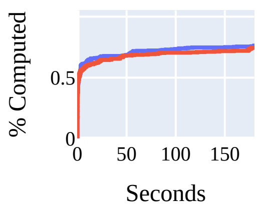
4.1 Feature computation time



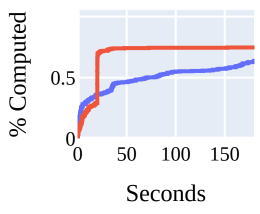
(a) Preliminary.



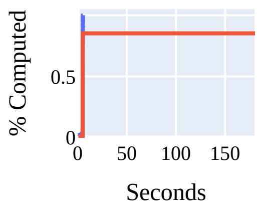
(b) Basic.



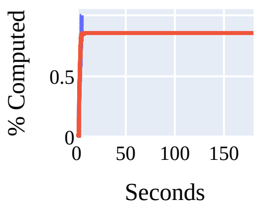
(c) KLB.



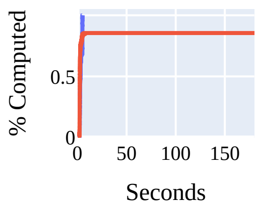
(d) Clause graph.



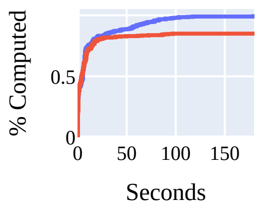
(e) Clause learning.



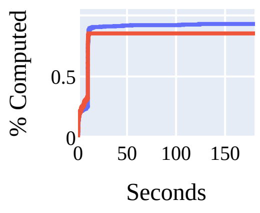
(f) LS: GSAT.



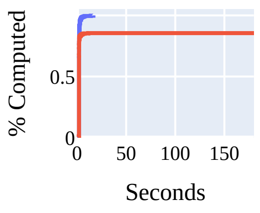
(g) LS: SAPS/Sparrow.



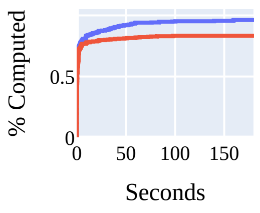
(h) Survey propagation.



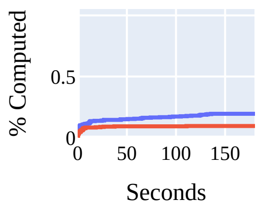
(i) Diameter.



(j) Lobjois.



(k) Unit propagation.



(l) Linear programming.

Figure 1 Percentage of features computed by the old tool (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue) over the available time budget for each feature group on the 2022 SAT Competition. For most feature groups, the new tool extracts features from more instances than the old one.

Figure 1 and Figure 2 show the percentage of features computed over the available time budget using the old (depicted in red) and new (depicted in blue) SATZilla tool on the 2022 and 2023 SAT Competition data, respectively. For simplicity, due to the similarity of the overall results between the two competitions, we draw more detailed insights based on plots from the 2023 edition (Figure 2).

We first observe that the new tool is able to extract more features than the old one for most feature groups. In particular, we highlight the performance gains on the preliminary feature group (Figure 2a), for which the new tool can extract the features for all formulas, compared to less than 80% of the formulas when using the the old tool. We note that for some feature groups, like graph learning (Figure 2d), the old tool is able to extract more features compared to the preliminary feature group. This is due to the fact that the old tool extracts the preliminary, basic, KLB and CG feature groups together. Therefore, in case computing one of those groups takes a long time, the whole feature extraction fails.

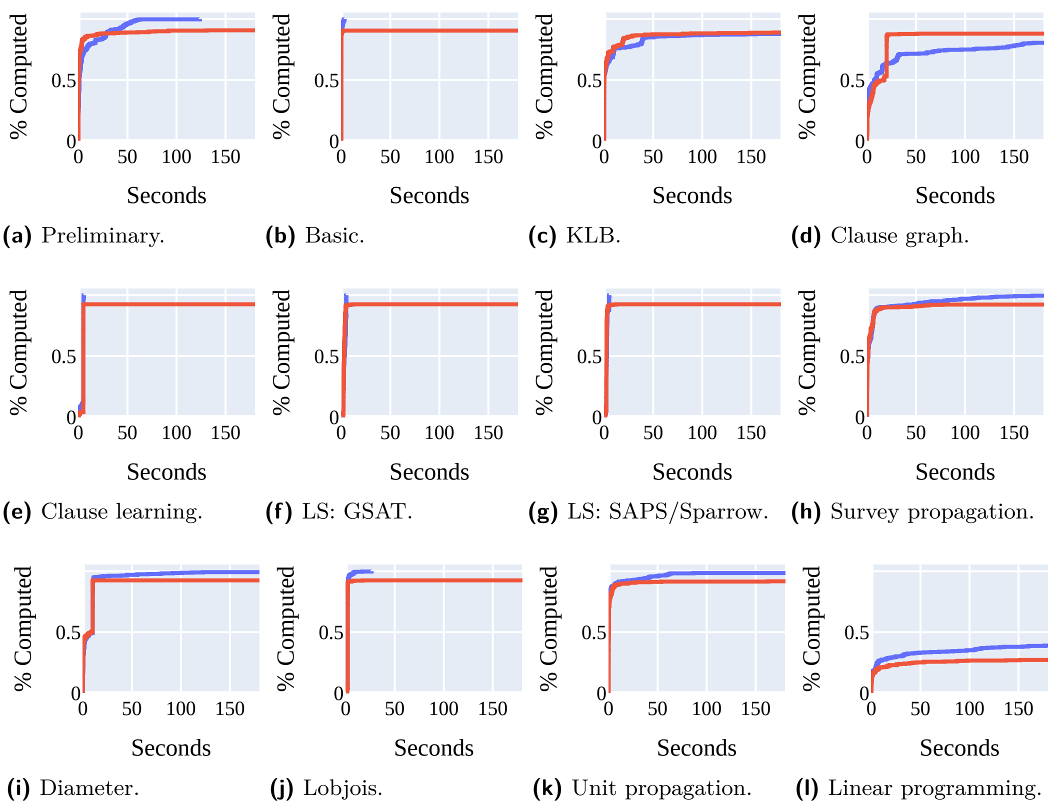


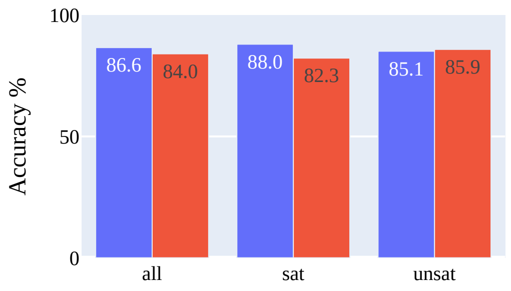
Figure 2 Percentage of features computed by the old tool (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue) over the available time budget for each feature group on the 2023 SAT Competition. For most feature groups, the new tool extracts features from more instances than the old one.

We also note that for KLB and CG features there is an advantage for the old version, due to the new SBVA preprocessing method yielding larger formulas than its predecessor SATELITE (as in many cases smaller formulas are not always easier to solve). Similarly, the expensive graph-based features (*e.g.*, Figure 2d and Figure 2i) require more time to extract than smaller formulas. This is more apparent in the 2022 SAT Competition, where larger instances were used than in the 2023 SAT Competition.

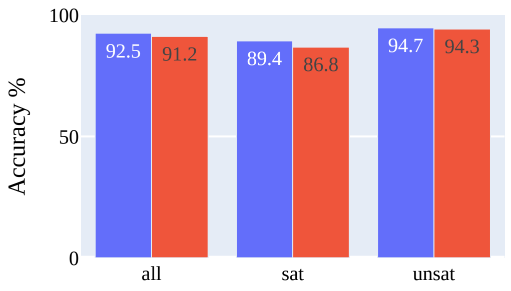
4.2 Satisfiability prediction

The first downstream task is satisfiability prediction, for which we measure the performance when using features extracted by our tool. We use the random forest (RF) classifier from the scikit-learn package to learn the mapping between features (representing SAT instances) and outputs (satisfiable or unsatisfiable). We optimise the hyperparameters of the RF for one hour using SMAC3 [9] on 10-fold cross-validation of the training data. We consider instances from the 2022 and 2023 SAT Competitions for which we know the satisfiability result (put differently, we omit instances for which the solution is unknown). On each competition, we evaluate the performance of the RF model using 10-fold cross-validation. This results in having outer cross-validation (for evaluation) and inner cross-validation (for training). Such techniques have been previously used by AutoFolio [10].

We present the satisfiability prediction accuracy scores in Figure 3. We see that, by using features extracted via the new tool, we achieve better performance across all instances on both SAT competitions. Furthermore, we notice that the new tool leads to a higher accuracy gain for satisfiable instances than for unsatisfiable instances. For the latter, the accuracy remains very similar to the one achieved by using the old tool in the 2023 SAT Competition and slightly dropped for the 2022 SAT Competition. This might be due to the fact that the unsatisfiable instances are larger on average, thus being more prone to timeouts even when using our new tool, which goes along with the worse performance in the 2022 SAT Competition, where the unsatisfiable instances were larger than in the 2023 SAT Competition. We point out that such high accuracy was already achieved before for industrial SAT instances [2].



(a) 2022 SAT Competition.



(b) 2023 SAT Competition.

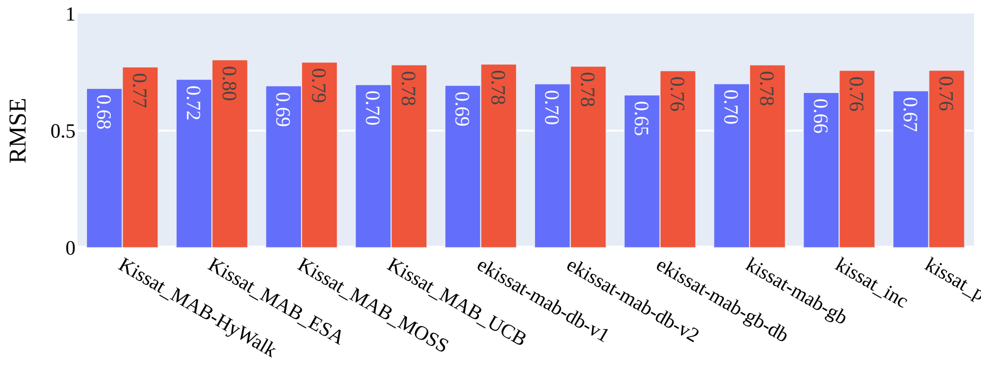
Figure 3 Accuracy of the satisfiability prediction task using a random forest with features extracted by the old (SATZilla 2012) and the new tool (SATZilla 2024). We see an overall higher accuracy for the new tool, which results from higher accuracy on satisfiable instances. On unsatisfiable ones, the accuracy remains the same.

4.3 Performance prediction

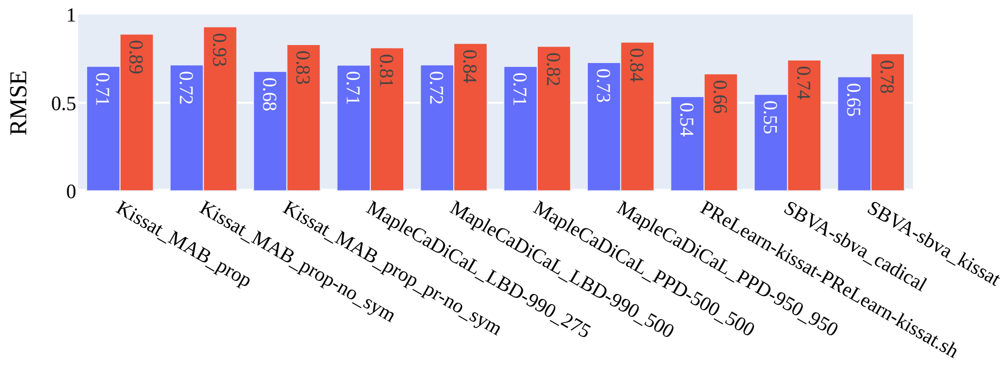
The second downstream task we investigate is performance prediction, which has important applications in algorithm selection, configuration and benchmark generation. We refer to the methodology described in [7] and use a RF regressor as the EPM. It is important to note that for an accurate running time prediction we need to perform a log_{10} transformation of running times prior to training the model, as done by Hutter *et al.* [7]. This transformation allows to capture the order of magnitude rather than small variations in the running time. The EPM then maps instance features to the log-transformed running times, and we aim to minimise the root mean squared error (RMSE) of the model, which is defined as $\sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} (\hat{y}_i - y_i)^2}$ for n predicted running times $\hat{y}_i$ and ground truth running times y_i. Lower RMSE scores are better and 0 is the optimal value. In line with the previous task, we perform inner and outer cross-validation and optimise the RF's hyperparameters for one hour using SMAC3.

We look into the performance of the EPM for running time prediction on all solvers from the 2022 and 2023 SAT Competitions. Here, we do not actually run solvers on instances to record their running times, but rather use the running times reported by the competitions. We display the results for the 10 best solvers from each competition in Figure 4 (the results for all solvers can be found in the supplementary material). We see that using the features extracted by the new tool leads to the lower RMSE for all solvers, compared to using those extracted by the old tool. For some solvers, we observe a significantly lower RMSE, like

Kissat_MAB_prop-no_sym, where using the new tool decreases the RMSE from 0.84 to 0.69. Figure 5 shows the histogram of the error percentage for all solvers in the 2022 and 2023 SAT Competitions. We see that for the 2022 competition, the new tool has more instances with error rate lower than 10% compared to the old version. For the 2023 competition, using the features extracted with the new version, more instances are predicted with less than 1% error. Histograms per solver are available in Appendix B.



(a) 2022 SAT Competition.



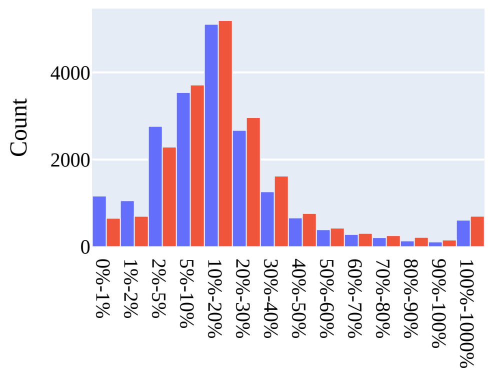
(b) 2023 SAT Competition.

Figure 4 Root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. The new tool achieves lower RMSE for all solvers.

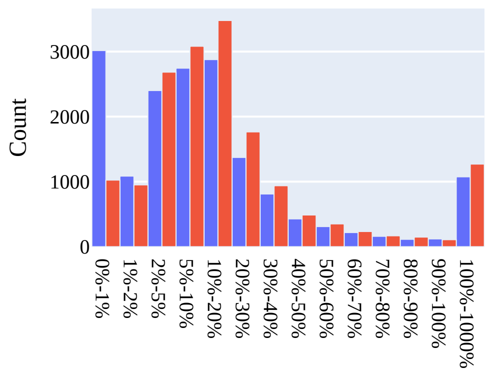
4.4 Algorithm selection

The third and final downstream task we consider is algorithm selection. In algorithm selection, given a set of instances I, a set of solvers (*i.e.*, algorithm portfolio) $\mathbf{A}$ and a performance metric $m : \mathbf{A} \times I \rightarrow \mathbb{R}$, we build an algorithm selector $S : I \rightarrow \mathbf{A}$ such that its performance is optimal on the instance set I according to the metric m. We compare the performance of algorithm selection to two standard baselines, the single best solver (SBS; *i.e*, the solver with the lowest overall running time) and the virtual best solver (VBS; *i.e.*, the theoretical oracle which for each instance selects the actual best solver on it).

We measure the performance of algorithm selection using *closed gap*, which is computed as $\frac{m_{SBS}-m_S}{m_{SBS}-m_{VBS}}$, (*i.e.*, the closed gap stands for how much of the gap between the SBS and the VBS is closed by using the algorithm selector). We then use AutoFolio [10] as an algorithm



(a) 2022 SAT Competition.



(b) 2023 SAT Competition.

Figure 5 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. The new tool achieves lower error percentages.

selector, which consists of multiple algorithm selection approaches, from which the best one is suggested by using algorithm configuration. As algorithm portfolio for the selection, we use the 10 best solvers from each SAT competition. We train the selector for 8 hours.

Figure 6 shows the closed gap results on the 2022 and 2023 SAT Competitions. Positive closed gap values on both scenarios using both the old and the new tool indicate that, in general, SATZilla features are useful for the algorithm selection task. Importantly, features extracted with the new tool lead to better closed gap values on both scenarios. In Figure 7, we provide ECDF plots showing the fraction of instances solved over time. In the 2022 SAT Competition scenario, the old version of the tool performs worse than the SBS until a budget of approximately 1000 seconds, while the new version of the tool performs similarly to the SBS. After 1000 seconds, both versions of the SATZilla features perform similarly. In the 2023 SAT Competition scenario, the new tool performs better than the old one for budgets between 100 and 1000 seconds. With a budget of less than 10 seconds, the old tool solves more instances. Overall, the ECDF plots reflect well what is shown in Figure 6, where we see that the new tool exhibits a few percents higher closed gap value.

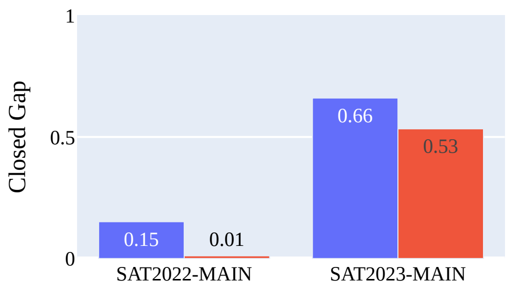
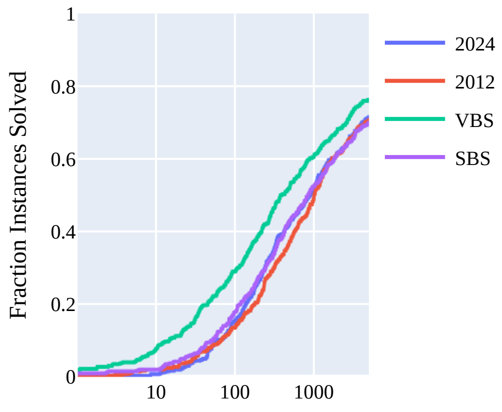
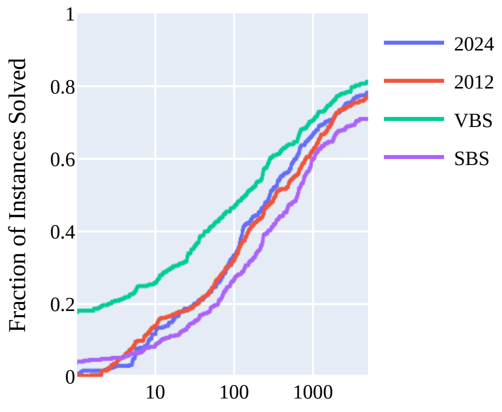


Figure 6 Closed gap values for the algorithm selection task using the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue) on the 2022 and 2023 SAT Competitions; higher is better.



(a) 2022 SAT Competition.



(b) 2023 SAT Competition.

Figure 7 ECDF plots for the algorithm selection task using the old (SATZilla 2012) and the new tool (SATZilla 2024) on the 2022 and 2023 SAT Competitions.

5 Conclusions

In this paper, we introduced an improved version of the well-known SATZilla feature extraction tool, motivated by the need to facilitate the feature extraction process by incorporating better user infrastructure and bug fixing. Our new version uses most up-to-date preprocessing techniques and SAT solvers, which allow for better representation of SAT formulas. Our experiments showed that, by using the new tool, we achieve a more accurate satisfiability prediction, a lower error for running time prediction, and a better closed gap for algorithm selection.

Our new SATZilla 2024 extraction tool aims to facilitate and promote the usage of SAT features even beyond their current scope. It is easily extensible and thus encourages building atop, *e.g.*, by looking into features based on the recent developments in the explainability of SAT solvers [3], or other advancements in SAT.

References

1 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*, pages 51–53, 2020. URL: `https://researchportal.helsinki.fi/en/publications/proceedings-of-sat-competition-2020-solver-and-benchmark-descript`.

2 David Devlin and Barry O'Sullivan. Satisfiability as a classification problem. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*, pages 1–10, 2008. URL: `http://www.cs.ucc.ie/~osullb/pubs/classification#:~:text=Sat%20can%20be%20seen%20as,the%20problem%20of%20deciding%20Sat`.

3 Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström, and Laurent Simon. Seeking practical CDCL insights from theoretical SAT benchmarks. In Jérôme Lang, editor, *Proceedings of the 37th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1300–1308, 2018. `doi:10.24963/ijcai.2018/181`.

4 Wenxuan Guo, Hui-Ling Zhen, Xijun Li, Wanqian Luo, Mingxuan Yuan, Yaohui Jin, and Junchi Yan. Machine learning methods in solving the boolean satisfiability problem. *Machine Intelligence Research*, 20(5):640–655, 2023. `doi:10.1007/s11633-022-1396-2`.

5 Eric I. Hsu and Sheila A. McIlraith. VARSAT: integrating novel probabilistic inference techniques with DPLL search. In Oliver Kullmann, editor, *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 377–390, 2009. `doi:10.1007/978-3-642-02777-2_35`.

6 Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 507–523, 2011. `doi:10.1007/978-3-642-25566-3_40`.

7 Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014. `doi:10.1016/j.artint.2013.10.003`.

8 Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):22:1–22:52, 2009. `doi:10.1145/1538902.1538906`.

9 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23:54:1–54:9, 2022. URL: `http://jmlr.org/papers/v23/21-0888.html`.

10 Marius Lindauer, Holger H. Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015. `doi:10.1613/jair.4726`.

11 Lionel Lobjois and Michel Lemaître. Branch and bound algorithm selection by performance prediction. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pages 353–358, 1998. URL: `http://www.aaai.org/Library/AAAI/1998/aaai98-050.php`.

12 Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient SAT solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, volume 3542, pages 360–375, 2004. `doi:10.1007/11527695_27`.

13 Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In *Proceedings of the 10TH International Conference on Principles and Practice of Constraint Programming*, pages 438–452, 2004. `doi:10.1007/978-3-540-30201-8_33`.

14 Jeroen Rook, Anna Latour, Holger Hoos, and Siegfried Nijssen. Caching in model counters: A journey through space and time. *Proceedings of the Workshop on Counting and Sampling*, 2021. URL: `https://ada.liacs.nl/papers/RooEtAl21.pdf`.

15 Olivier Roussel. Controlling a solver execution with the runsolver tool. *Journal Satisfiability Boolean Modelling Computation*, 7(4):139–144, 2011. `doi:10.3233/sat190083`.

16 Hadar Shavit and Holger Hoos. hadarshavit/revisiting_satzilla. Software, swhId: `swh:1:dir:a06a5e5ad5473cd07b6a423a1dc5491b54af7c61` (visited on 2024-07-04). URL: `https://github.com/hadarshavit/revisiting_satzilla`.

17 Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. `doi:10.1613/jair.2490`.

18 Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge 2012*, pages 57–58, 2012. URL: `https://www.cs.ubc.ca/~kevinlb/papers/2012-SATzilla2012-Solver-Description.pdf`.

A Running time prediction results

In this appendix, we present the full results of running time prediction for all solvers from the 2022 and 2023 SAT Competitions. In Table 1 and Table 2 we show the results for the 2022 SAT Competition, where we see that our new tool constantly outperforms the old one. Similarly, in Table 3 we show the results for the 2023 SAT Competition, where using the features extracted by our new tool leads to lower RMSE than with the old one.

Table 1 RMSE of random forest for predicting log-transformed running times of SAT solvers from the 2022 SAT Competition using the old and new SATZilla features.

Solver	SATZilla 2012	SATZilla 2024
CaDiCaL-watchsat-lto	0.70	**0.63**
CaDiCaL_DVDL_V1	0.73	**0.65**
CaDiCaL_DVDL_V2	0.75	**0.65**
CadicalReorder	0.75	**0.66**
Cadical_ESA	0.75	**0.66**
IsaSAT	0.73	**0.64**
Kissat-MAB-rephasing	0.74	**0.67**
Kissat_MAB-HyWalk	0.77	**0.68**
Kissat_MAB_ESA	0.80	**0.72**
Kissat_MAB_MOSS	0.79	**0.69**
Kissat_MAB_UCB	0.78	**0.70**
Kissat_adaptive_restart	0.72	**0.68**
Kissat_cfexp	0.79	**0.71**
LSTech_CaDiCaL	0.77	**0.63**
LSTech_Maple	0.74	**0.66**
LSTech_kissat	0.76	**0.67**
LStech-Maple-BandSAT	0.67	**0.62**
LStech-Maple-FPS	0.74	**0.65**
LStech-Maple-HyWalk	0.72	**0.64**
MapleLCMDistChrBt-DL-v3	0.61	**0.58**
MergeSat 4.0-rc-rc3	0.65	**0.61**
SLIME SC-2022	0.70	**0.65**
SLIME SC-2022-alpha	0.70	**0.66**
SLIME SC-2022-beta	0.71	**0.66**
SLIME SC-2022-gamma	0.73	**0.69**
SeqFROST-ERE-All	0.76	**0.68**
SeqFROST-NoExtend	0.77	**0.67**
cadical-hack-gb	0.73	**0.62**
cadical_rel_Scavel	0.71	**0.61**

Table 2 RMSE of random forest for predicting log-transformed running times of SAT solvers from the 2022 SAT Competition using the old and new SATZilla features (contd.).

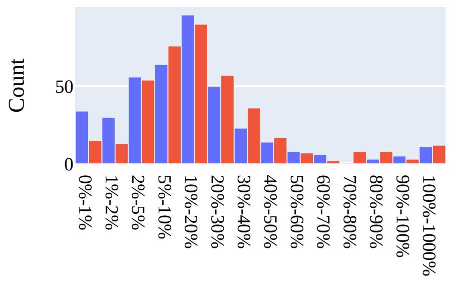
Solver	SATZilla 2012	SATZilla 2024
ekissat-mab-db-v1	0.78	**0.69**
ekissat-mab-db-v2	0.78	**0.70**
ekissat-mab-gb-db	0.76	**0.65**
glucose-reboot	0.78	**0.68**
hCaD_V1-psids	0.69	**0.63**
hCaD_V2	0.68	**0.64**
hKis-psids	0.69	**0.59**
hKis-sat	0.79	**0.69**
hKis-unsat	0.74	**0.66**
kissat-els-v1	0.76	**0.67**
kissat-els-v2	0.74	**0.67**
kissat-els-v3	0.78	**0.67**
kissat-els-v4	0.77	**0.66**
kissat-mab-gb	0.78	**0.70**
kissat-sc2022-bulky	0.76	**0.69**
kissat-sc2022-hyper	0.76	**0.70**
kissat-sc2022-light	0.75	**0.69**
kissat-watchsat-lto	0.73	**0.67**
kissat_inc	0.76	**0.66**
kissat_pre	0.76	**0.67**
kissat_relaxed	0.72	**0.67**

Table 3 RMSE of random forest for predicting log-transformed running times of SAT solvers from the 2023 SAT Competition using the old and new SATZilla features.

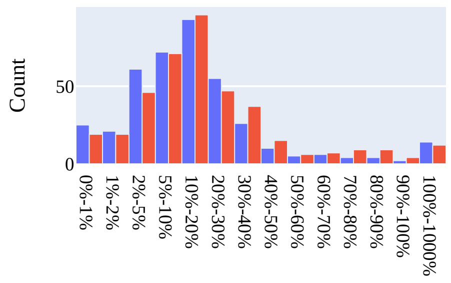
Solver	SATZilla 2012	SATZilla 2024
AMSAT_	0.80	**0.77**
BreakID-kissat-low.sh	0.92	**0.83**
CaDiCaL_vivinst	0.84	**0.73**
Cadical_ESA	0.85	**0.74**
Cadical_rel_1.5.3.Scavel	0.80	**0.74**
IsaSAT	0.90	**0.82**
Kissat_Inc_ESA	0.88	**0.73**
Kissat_MAB_Binary	0.89	**0.72**
Kissat_MAB_Conflict	0.87	**0.72**
Kissat_MAB_Conflict+	0.88	**0.75**
Kissat_MAB_DeepWalk+	0.87	**0.73**
Kissat_MAB_ESA	0.91	**0.74**
Kissat_MAB_Rephases	0.84	**0.71**
Kissat_MAB_prop	0.89	**0.71**
Kissat_MAB_prop-no_sym	0.93	**0.72**
Kissat_MAB_prop_pr-no_sym	0.83	**0.68**
MapleCaDiCaL_LBD-990_275	0.81	**0.71**
MapleCaDiCaL_LBD-990_500	0.84	**0.72**
MapleCaDiCaL_PPD-500_500	0.82	**0.71**
MapleCaDiCaL_PPD-950_950	0.84	**0.73**
MergeSat-bve_gates	0.75	**0.71**
MergeSat-bve_semgates	0.74	**0.73**
MergeSat-thread1	0.68	**0.67**
MiniSat+XorEngine	0.79	**0.77**
PReLearn-kissat-PReLearn-kissat.sh	0.66	**0.54**
PReLearn-kissat-PReLearn-tern-kissat.sh	0.55	**0.46**
ReEncode-kissat-ReEncode-pair-kissat.sh	0.75	**0.68**
SBVA-sbva_cadical	0.74	**0.55**
SBVA-sbva_kissat	0.78	**0.65**
SeqFROST	0.81	**0.73**
SeqFROST-ERE-All	0.81	**0.71**
SeqFROST-NoExtend	0.79	**0.74**
hKis-psids	0.81	**0.72**
hKis-sat_psids	0.80	**0.79**
hKis-unsat	0.86	**0.69**
hKissatInc-unsat	0.85	**0.77**
kissat-3.1.0	0.88	**0.76**
kissat-hywalk-exp	0.85	**0.70**
kissat-hywalk-exp-gb	0.89	**0.74**
kissat-hywalk-gb	0.88	**0.72**
kissat_incsp	0.95	**0.80**
tabularasat-1.0.0	0.88	**0.76**

B Running time prediction histograms

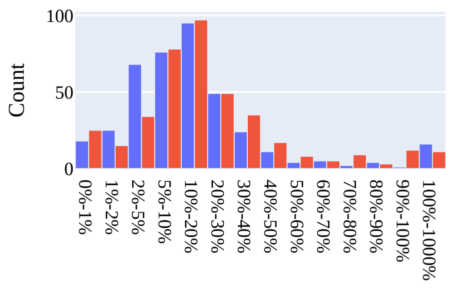
We present the histograms for the RMSE values per solver for running time prediction in Figures 8–19.



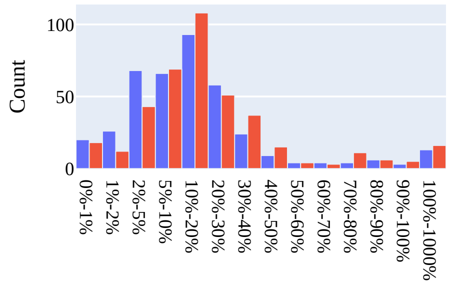
(a) 2022 Comp. CaDiCaL-watchsat-lto.



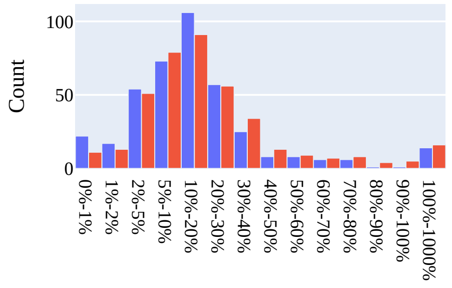
(b) 2022 Comp. CaDiCaL_DVDL_V1.



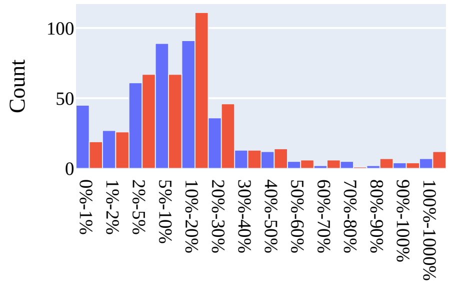
(c) 2022 Comp. CaDiCaL_DVDL_V2.



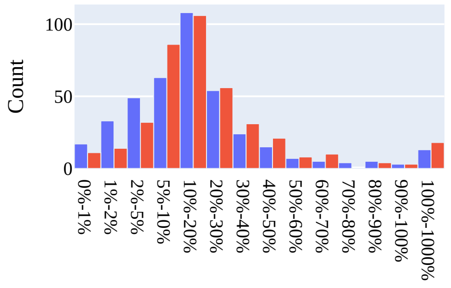
(d) 2022 Comp. CadicalReorder.



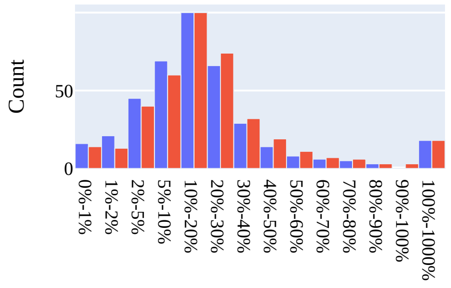
(e) 2022 Comp. Cadical_ESA.



(f) 2022 Comp. IsaSAT.

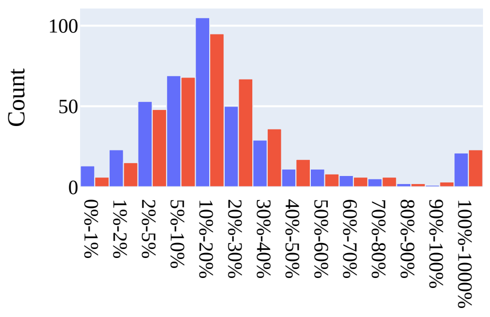


(g) 2022 Comp. Kissat-MAB-rephasing.

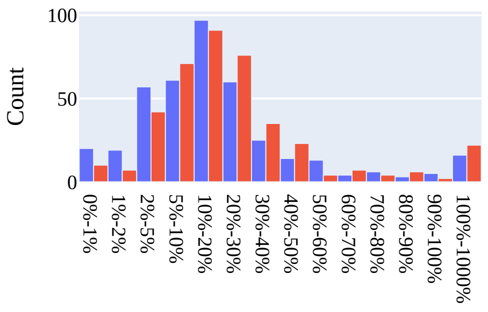


(h) 2022 Comp. Kissat_MAB-HyWalk.

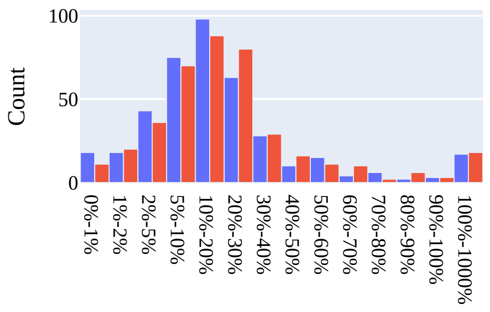
Figure 8 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver.



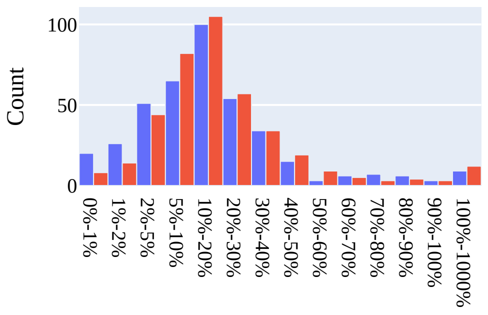
(a) 2022 Comp. Kissat_MAB_ESA.



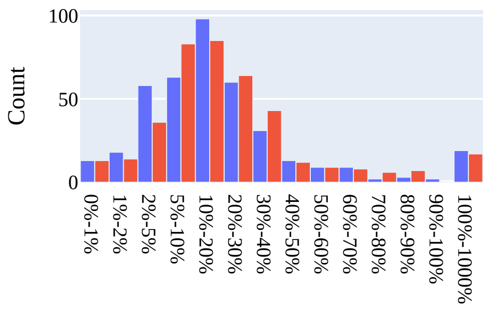
(b) 2022 Comp. Kissat_MAB_MOSS.



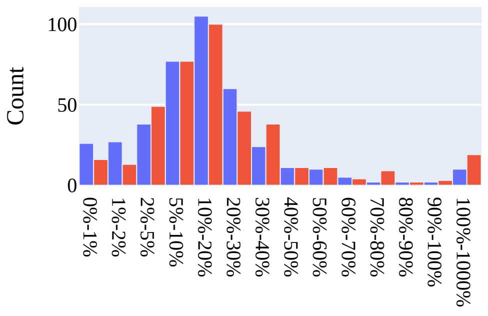
(c) 2022 Comp. Kissat_MAB_UCB.



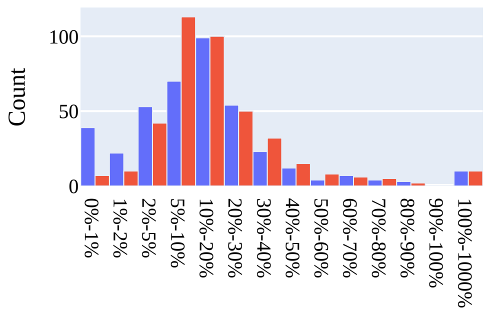
(d) 2022 Comp. Kissat_adaptive_restart.



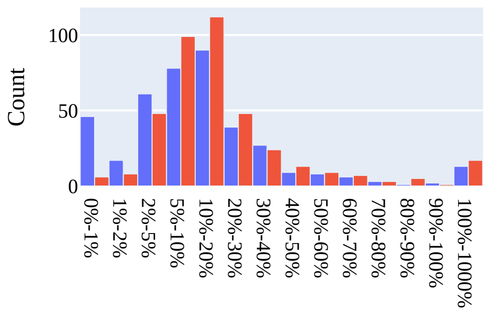
(e) 2022 Comp. Kissat_cfexp.



(f) 2022 Comp. LSTech_CaDiCaL.

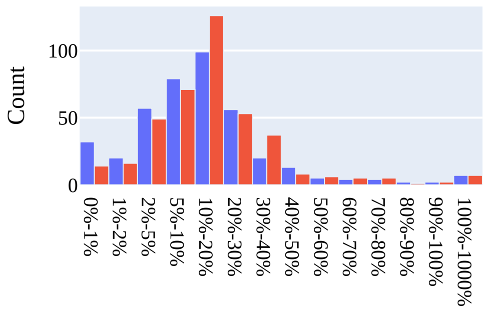


(g) 2022 Comp. LSTech_Maple.

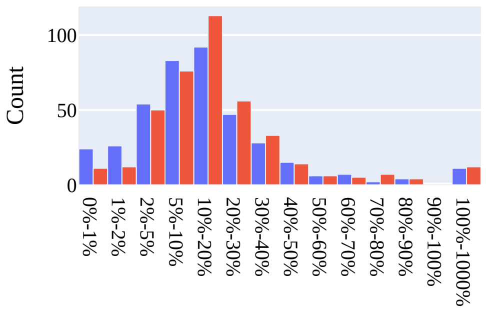


(h) 2022 Comp. LSTech_kissat.

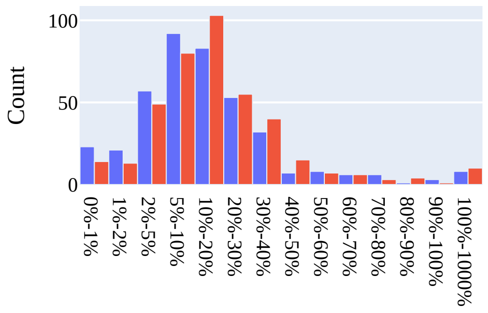
Figure 9 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



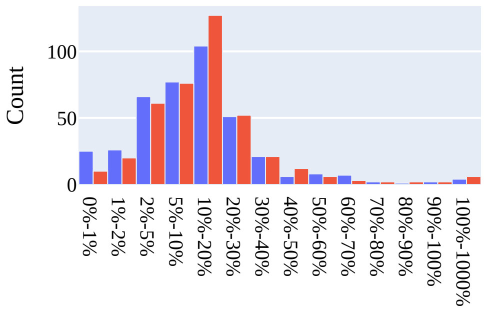
(a) 2022 Comp. LStech-Maple-BandSAT.



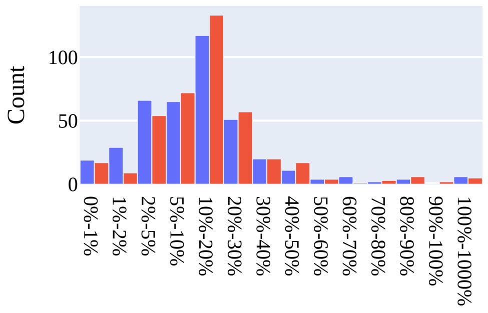
(b) 2022 Comp. LStech-Maple-FPS.



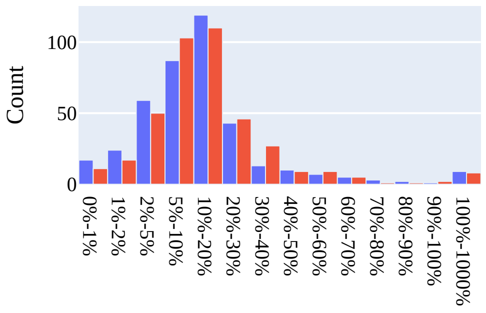
(c) 2022 Comp. LStech-Maple-HyWalk.



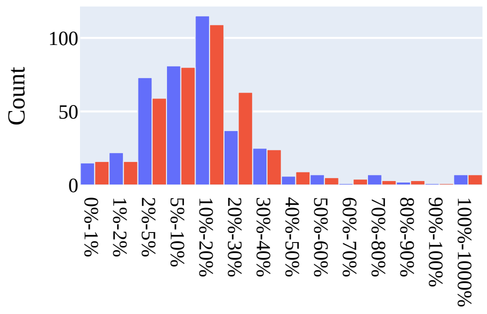
(d) 2022 Comp. MapleLCMDistChrBt-DL-v3.



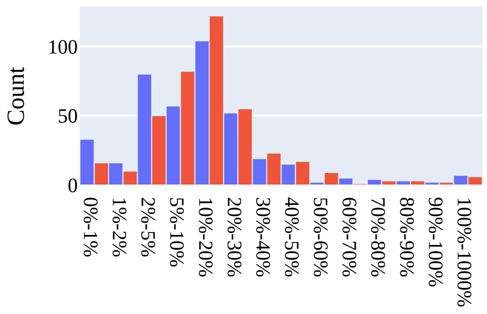
(e) 2022 Comp. MergeSat 4.0-rc-rc3.



(f) 2022 Comp. SLIME SC-2022.

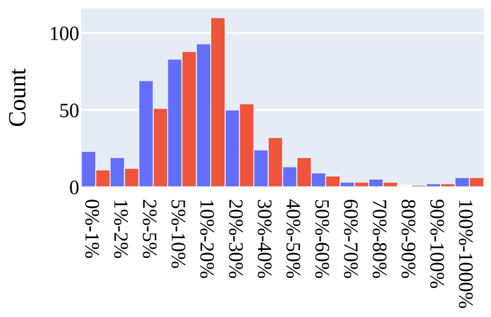


(g) 2022 Comp. SLIME SC-2022-alpha.

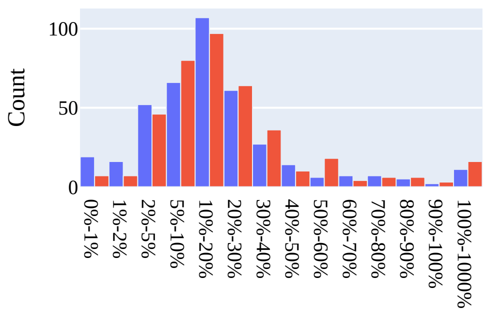


(h) 2022 Comp. SLIME SC-2022-beta.

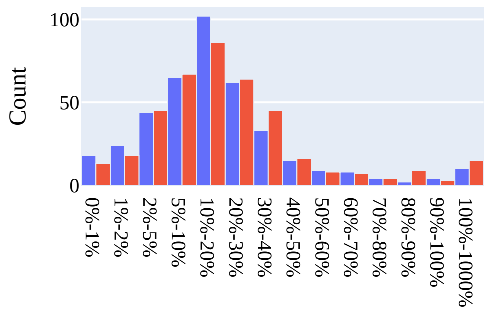
Figure 10 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



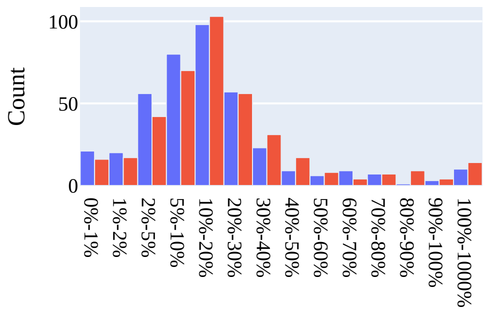
(a) 2022 Comp. SLIME SC-2022-gamma.



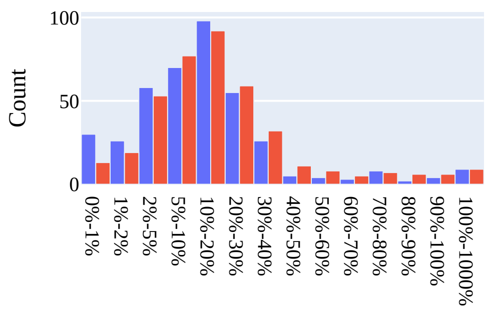
(b) 2022 Comp. SeqFROST-ERE-All.



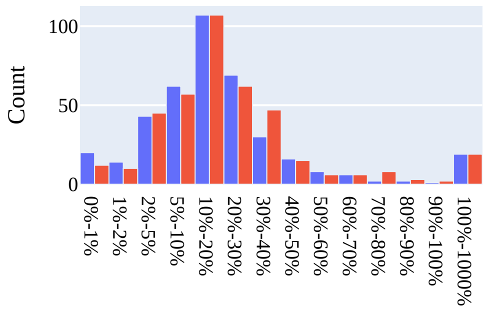
(c) 2022 Comp. SeqFROST-NoExtend.



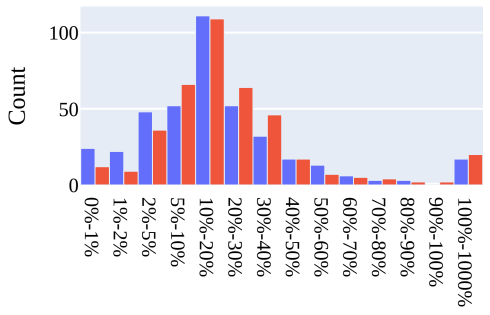
(d) 2022 Comp. cadical-hack-gb.



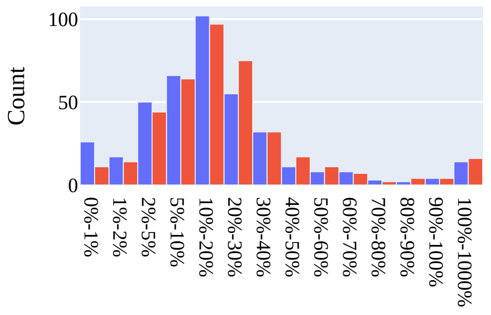
(e) 2022 Comp. cadical_rel_Scavel.



(f) 2022 Comp. ekissat-mab-db-v1.

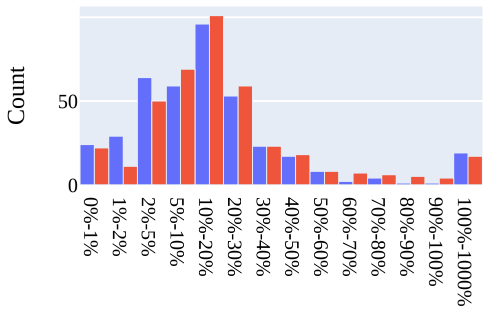


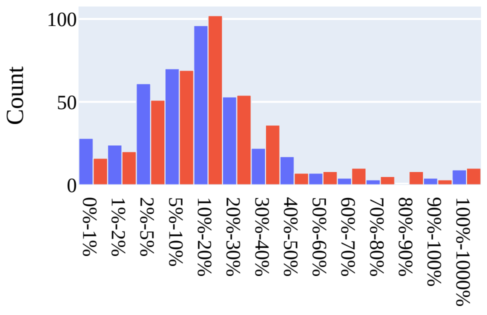
(g) 2022 Comp. ekissat-mab-db-v2.

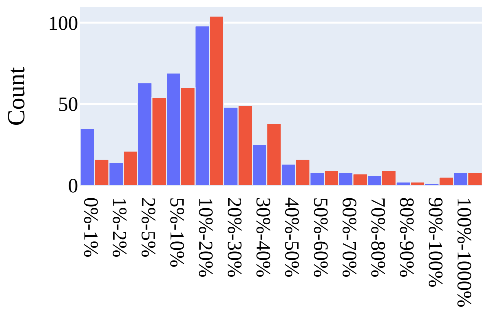


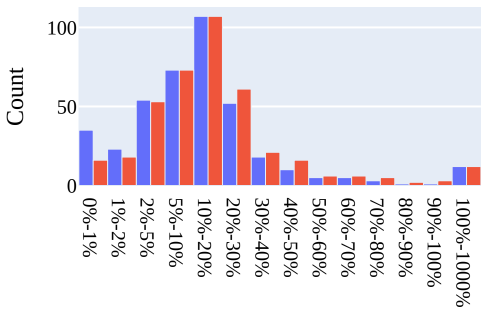
(h) 2022 Comp. ekissat-mab-gb-db.

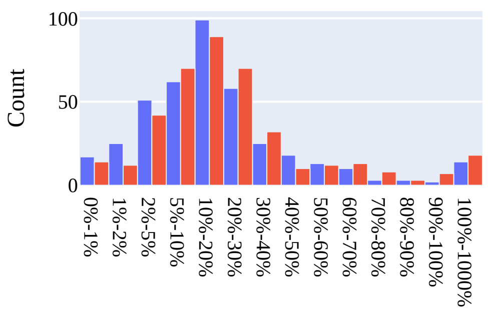
Figure 11 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).

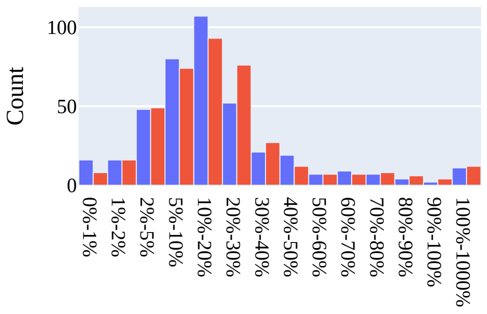

(a) 2022 Comp. glucose-reboot.

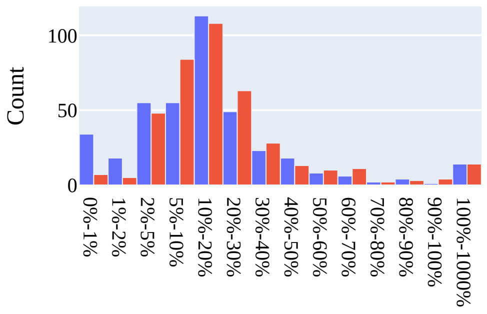

(b) 2022 Comp. hCaD_V1-psids.


(c) 2022 Comp. hCaD_V2.


(d) 2022 Comp. hKis-psids.


(e) 2022 Comp. hKis-sat.


(f) 2022 Comp. hKis-unsat.


(g) 2022 Comp. kissat-els-v1.

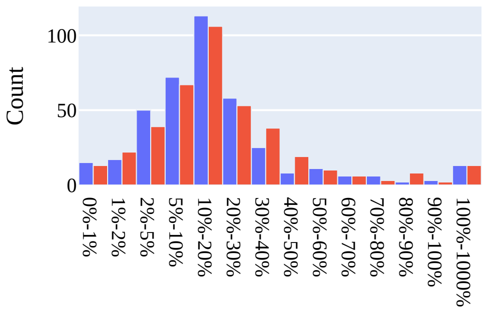
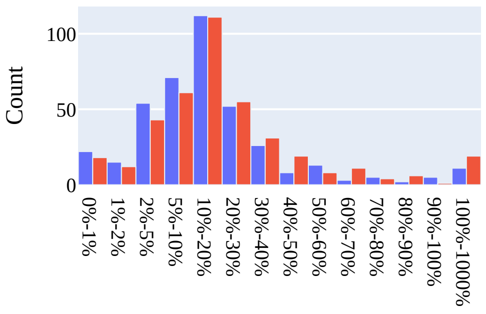
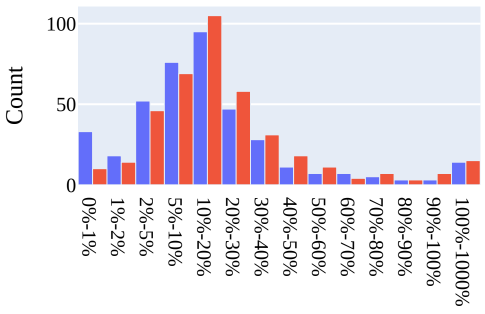

(h) 2022 Comp. kissat-els-v2.

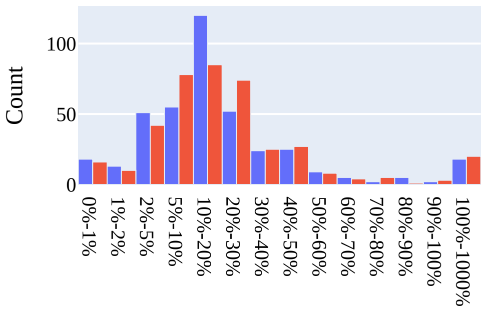
Figure 12 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



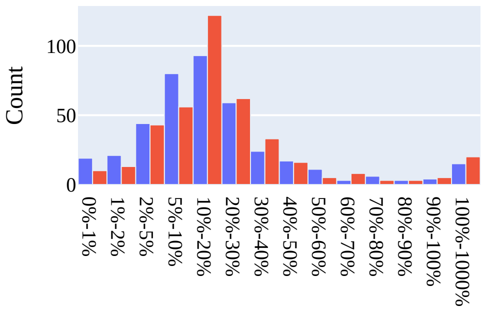
(a) 2022 Comp. kissat-els-v3.



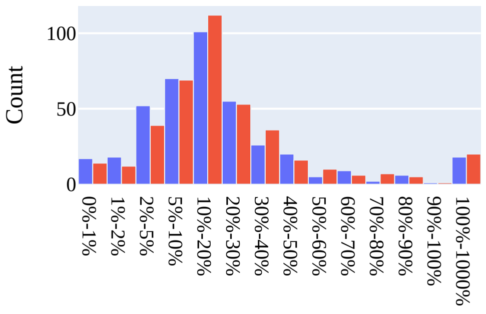
(b) 2022 Comp. kissat-els-v4.



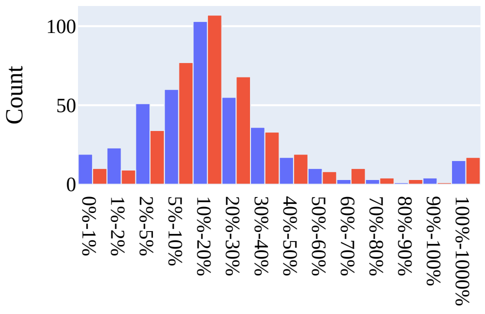
(c) 2022 Comp. kissat-mab-gb.



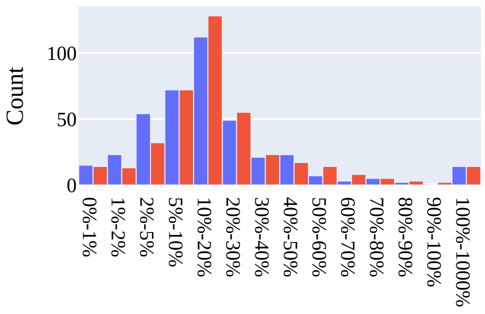
(d) 2022 Comp. kissat-sc2022-bulky.



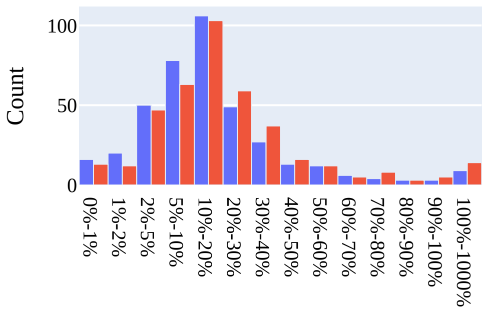
(e) 2022 Comp. kissat-sc2022-hyper.



(f) 2022 Comp. kissat-sc2022-light.

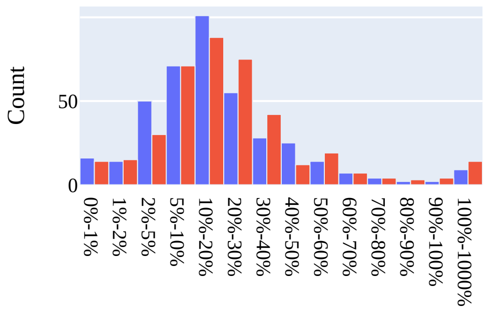


(g) 2022 Comp. kissat-watchsat-lto.

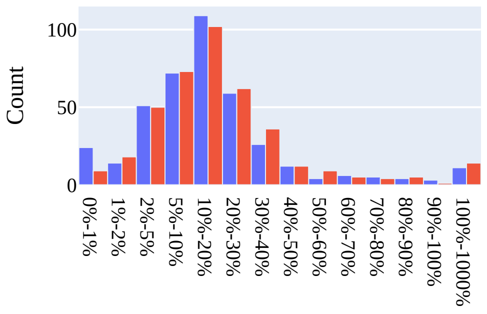


(h) 2022 Comp. kissat_inc.

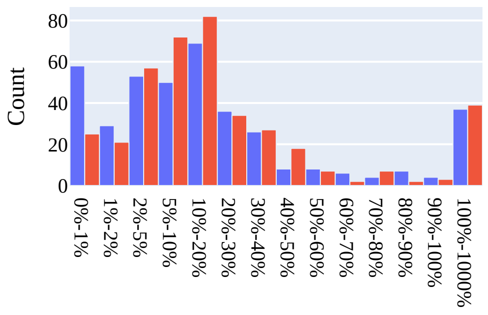
Figure 13 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



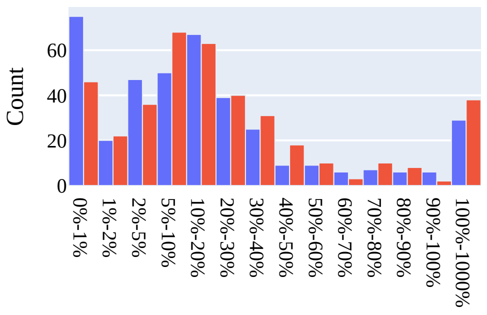
(a) 2022 Comp. kissat_pre.



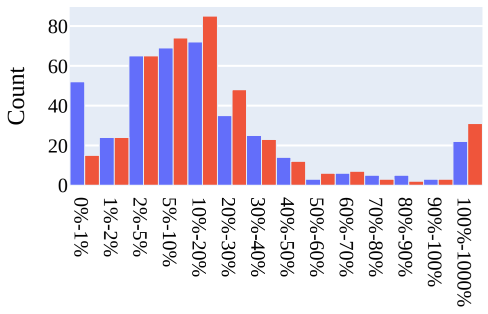
(b) 2022 Comp. kissat_relaxed.



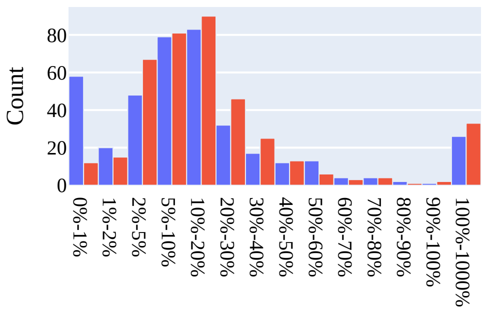
(c) 2023 Comp. AMSAT_.



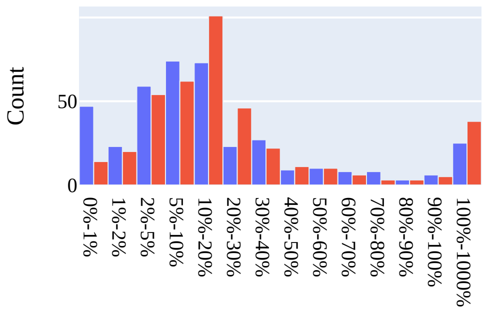
(d) 2023 Comp. BreakID-kissat-low.sh



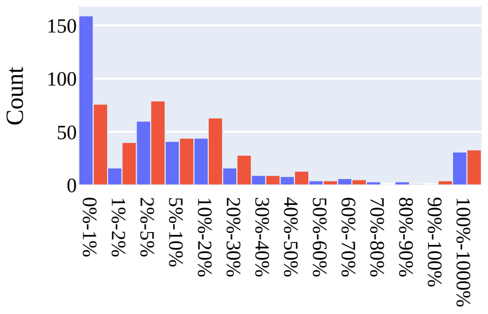
(e) 2023 Comp. CaDiCaL_vivinst.



(f) 2023 Comp. Cadical_ESA.

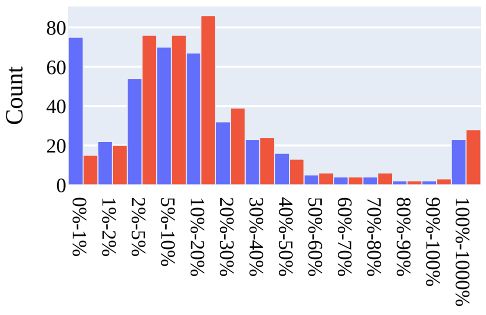


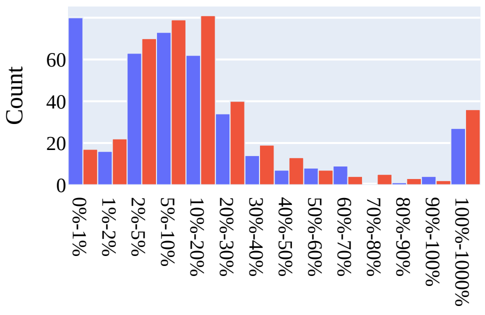
(g) 2023 Comp. Cadical_rel_1.5.3.Scavel.

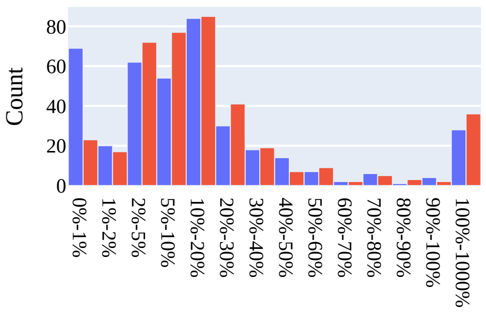


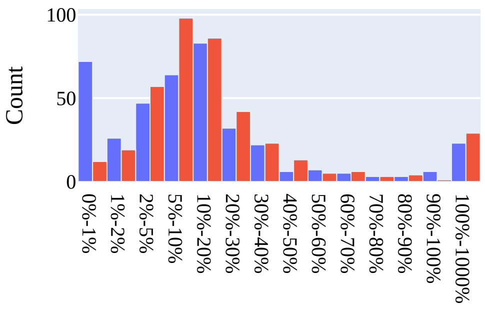
(h) 2023 Comp. IsaSAT.

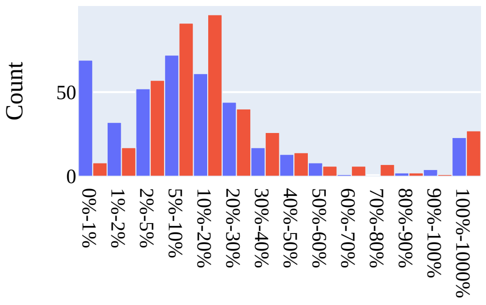
Figure 14 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).

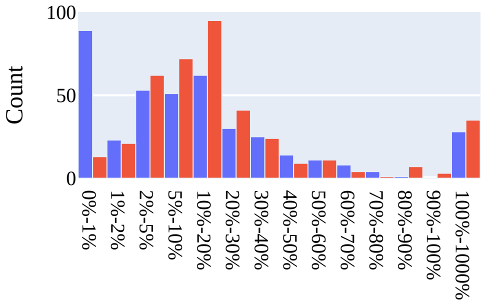

(a) 2023 Comp. Kissat_Inc_ESA.

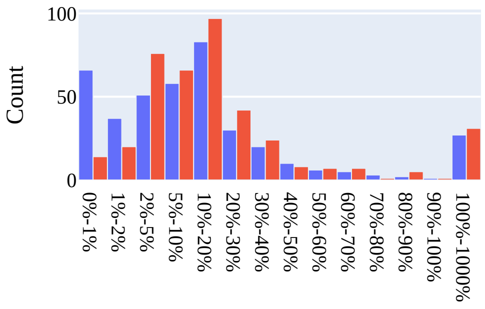

(b) 2023 Comp. Kissat_MAB_Binary.


(c) 2023 Comp. Kissat_MAB_Conflict.


(d) 2023 Comp. Kissat_MAB_Conflict+.


(e) 2023 Comp. Kissat_MAB_DeepWalk+.


(f) 2023 Comp. Kissat_MAB_ESA.


(g) 2023 Comp. Kissat_MAB_Rephases.

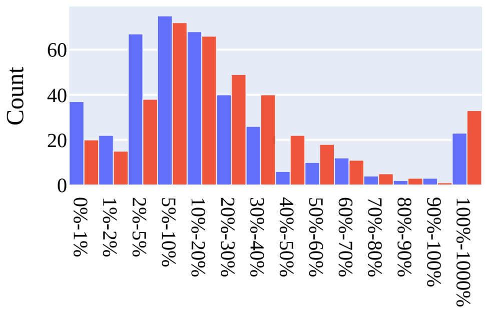
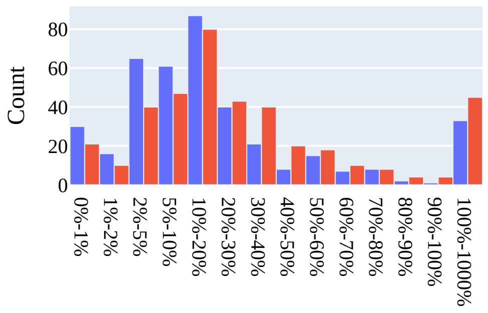
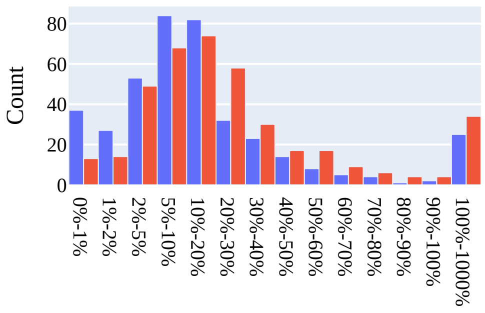

(h) 2023 Comp. Kissat_MAB_prop.

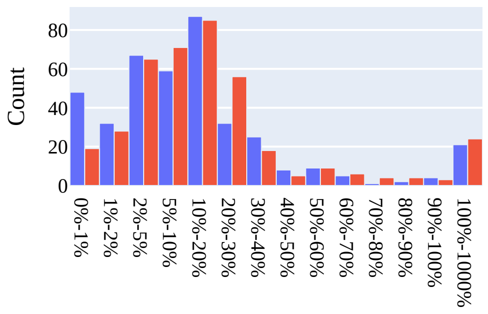
Figure 15 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



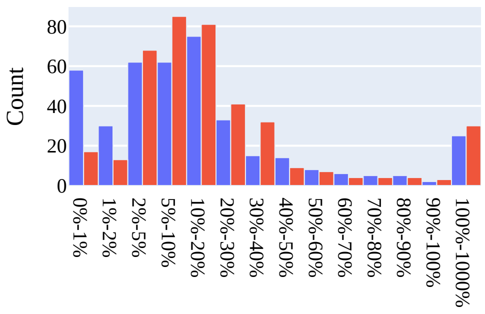
(a) 2023 Comp. Kissat_MAB_prop-no_sym.



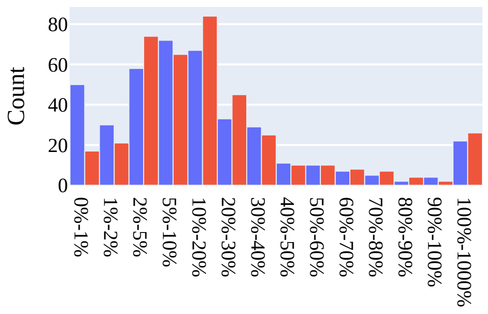
(b) 2023 Comp. Kissat_MAB_prop_pr-no_sym.



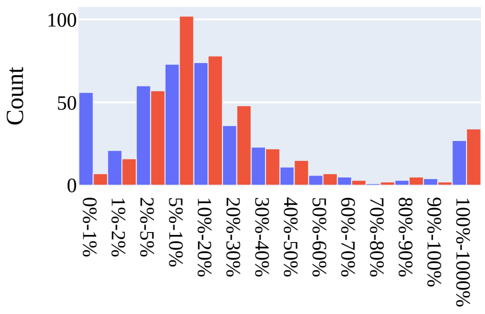
(c) 2023 Comp. MapleCaDiCaL_LBD-990_275.



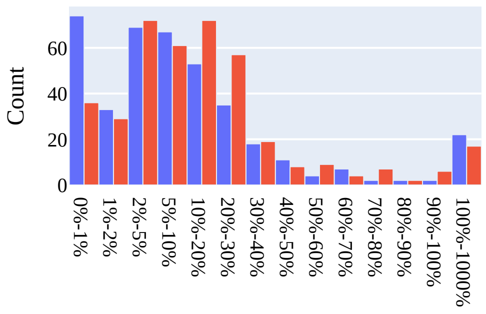
(d) 2023 Comp. MapleCaDiCaL_LBD-990_500.



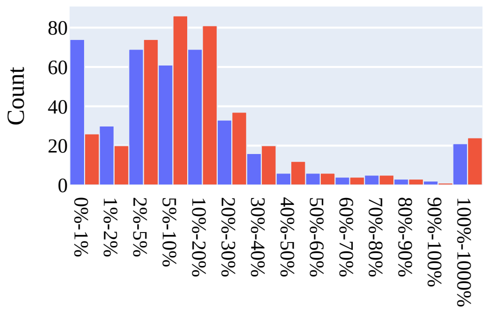
(e) 2023 Comp. MapleCaDiCaL_PPD-500_500.



(f) 2023 Comp. MapleCaDiCaL_PPD-950_950.

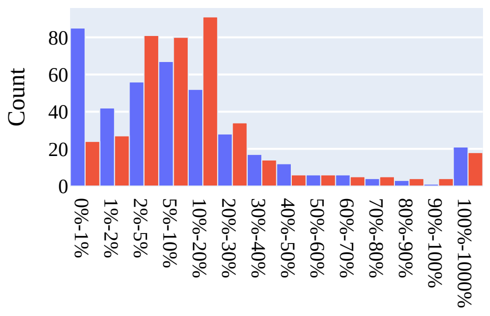


(g) 2023 Comp. MergeSat-bve_gates.

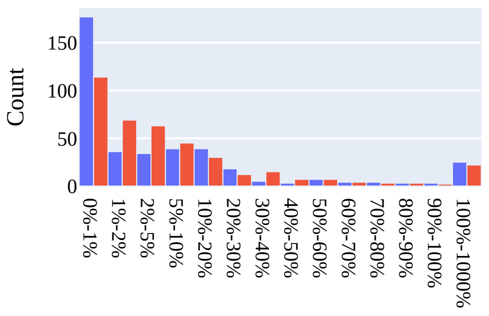


(h) 2023 Comp. MergeSat-bve_semgates.

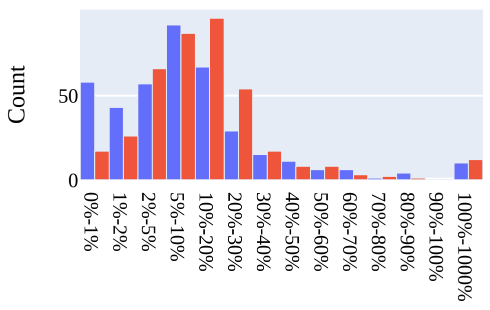
Figure 16 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



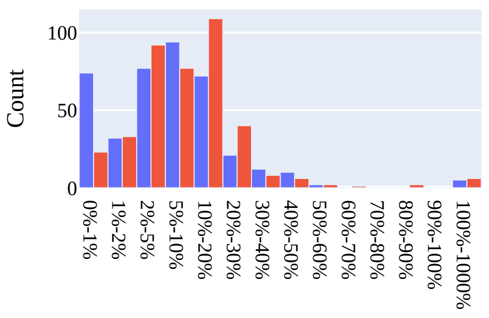
(a) 2023 Comp. MergeSat-thread1.



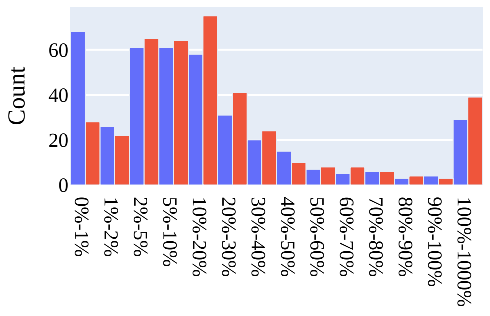
(b) 2023 Comp. MiniSat+XorEngine.



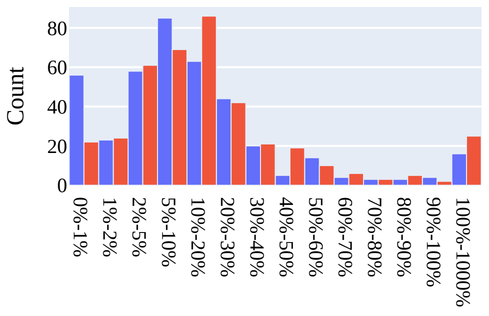
(c) 2023 Comp. PReLearn-kissat-PReLearn-kissat.sh.



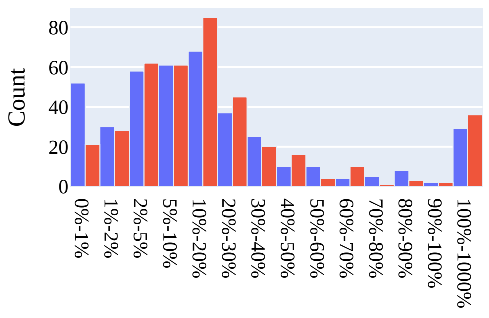
(d) 2023 Comp. PReLearn-kissat-PReLearn-tern-kissat.sh.



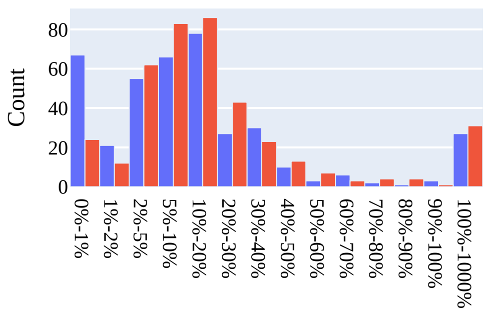
(e) 2023 Comp. ReEncode-kissat-ReEncode-pair-kissat.sh.



(f) 2023 Comp. SBVA-sbva_cadical.

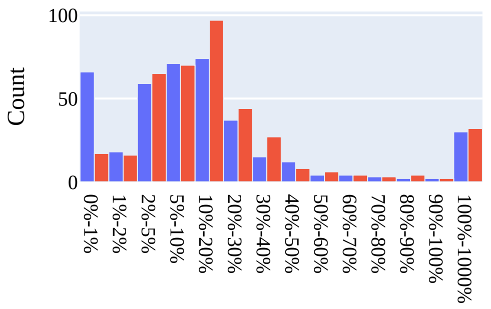


(g) 2023 Comp. SBVA-sbva_kissat.

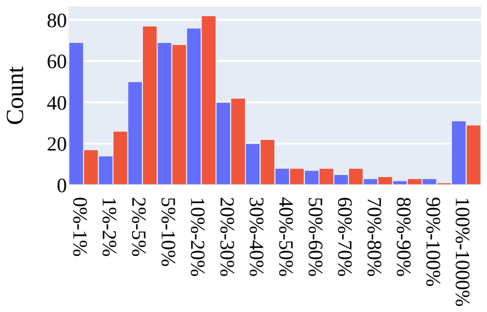


(h) 2023 Comp. SeqFROST.

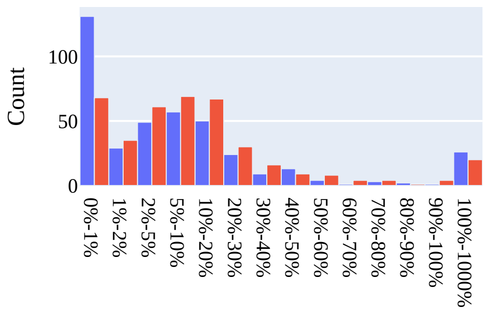
Figure 17 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



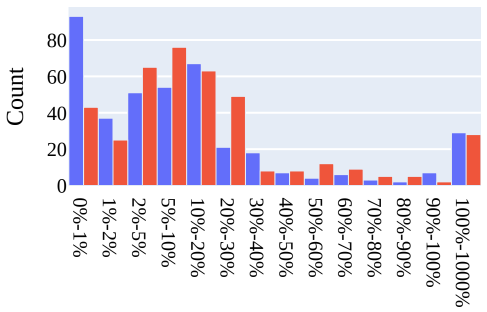
(a) 2023 Comp. SeqFROST-ERE-All.



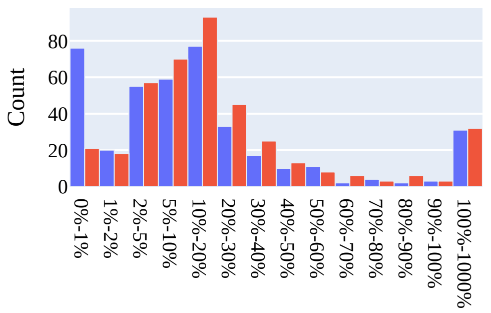
(b) 2023 Comp. SeqFROST-NoExtend.



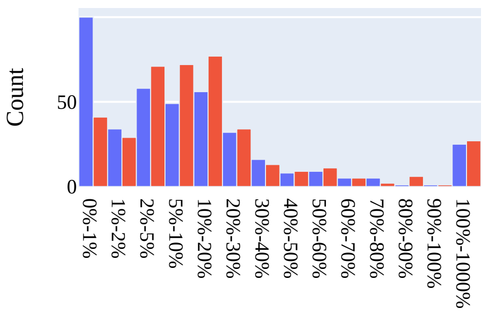
(c) 2023 Comp. hKis-psids.



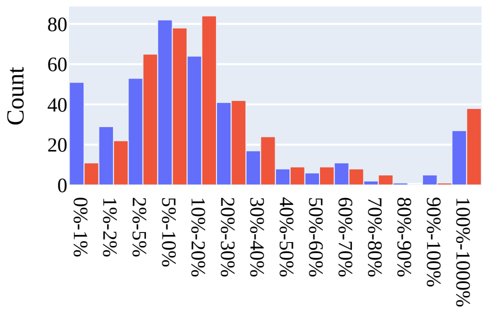
(d) 2023 Comp. hKis-sat_psids.



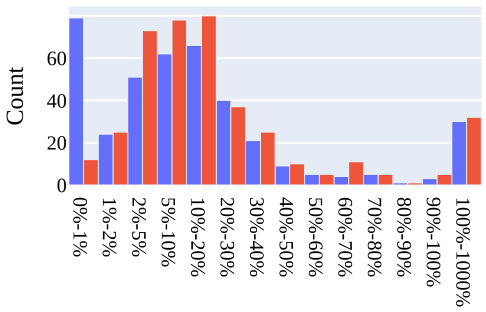
(e) 2023 Comp. hKis-unsat.



(f) 2023 Comp. hKissatInc-unsat.

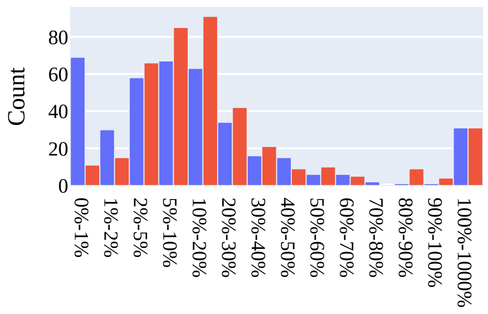


(g) 2023 Comp. kissat-3.1.0.

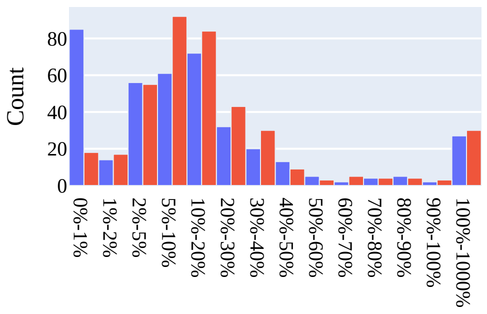


(h) 2023 Comp. kissat-hywalk-exp.

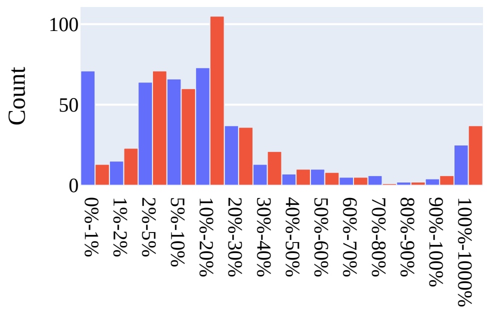
Figure 18 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).



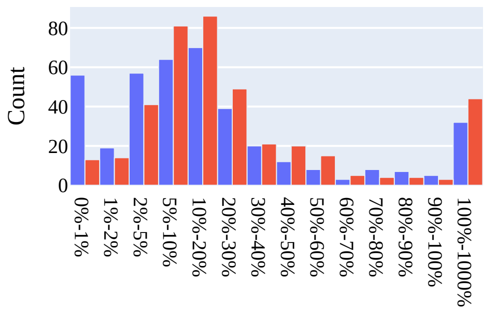
(a) 2023 Comp. kissat-hywalk-exp-gb.



(b) 2023 Comp. kissat-hywalk-gb.



(c) 2023 Comp. kissat_incsp.



(d) 2023 Comp. tabularasat-1.0.0.

Figure 19 Histogram of the error percentage of the root mean square error (RMSE) of (log-transformed) running time prediction using a random forest with features extracted by the old (SATZilla 2012; in red) and the new tool (SATZilla 2024; in blue), on SAT solvers from the 2022 and 2023 SAT Competitions. Results are presented per solver (contd.).

Strategy Extraction by Interpolation

Friedrich Slivovsky
Department of Computer Science, University of Liverpool, UK

Abstract

In applications, QBF solvers are often required to generate strategies. This typically involves a process known as strategy extraction, where a Boolean circuit encoding a strategy is computed from a proof. It has previously been observed that Craig interpolation in propositional logic can be seen as a special case of QBF strategy extraction. In this paper we explore this connection further and show that, conversely, any strategy for a false QBF corresponds to a sequence of interpolants in its complete (Herbrand) expansion. Inspired by this correspondence, we present a new strategy extraction algorithm for the expansion-based proof system Exp+Res. Its asymptotic running time matches the best known bound of $O(mn)$ for a proof with m lines and n universally quantified variables. We report on experiments comparing this algorithm with a strategy extraction algorithm based on combining partial strategies, as well as with round-based strategy extraction.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases QBF, Expansion, Strategy Extraction, Interpolation

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.28

Supplementary Material *Software (Source Code)*: `https://github.com/fslivovsky/ferpmodels`
archived at `swh:1:dir:ecf767c252fb45b1fbd12ebf83088d07c9025697`

Funding Vienna Science and Technology Fund (WWTF) grant 10.47379/ICT19060.

1 Introduction

Due to continuous performance improvements over the last 30 years [8], SAT solvers have become a standard tool in formal methods and electronic design automation [16, 39]. However, the increasing complexity of specifications in these areas can lead to prohibitively large encodings that are unmanageable even for the most efficient SAT solvers. This problem has prompted research into more succinct logics, such as Quantified Boolean Formulas (QBF), that can naturally encode a wide range of synthesis tasks [10, 11, 35, 37].

In many of these applications, QBF solvers cannot just answer "true" or "false", they are expected to provide a winning strategy as a solution. This typically involves *strategy extraction*, where a Boolean circuit encoding a strategy is computed from a proof generated by the solver. Determining whether a QBF proof system has efficient strategy extraction is thus important for practical concerns. But improved strategy extraction can also serve a tighter characterisation of proof systems. A seminal result in this context is linear-time strategy extraction for Q-resolution [2]: one can show that the extracted strategies are decision lists, and this leads to strong lower bounds against Q-resolution [4, 5].

Q-resolution is the proof system underpinning quantified CDCL, one of the main paradigms in QBF solving. Another main paradigm is counter-example guided expansion [23], with Exp+Res as its underlying proof system [24]. It has been shown that an Exp+Res refutation of a QBF can guide the universal player to win the evaluation game, and since all operations can be implemented in polynomial time, it follows that Exp+Res has polynomial-time strategy extraction [5].

However, the simulation of player moves in the resulting *round-based* strategy extraction algorithm incurs a significant overhead. In experiments, an implementation of this idea struggled to generate strategies for many QBFs that could be solved quickly [19]. An alternative is to simulate an operation for combining partial strategies [38] using circuits, which leads to strategy extraction in time $O(mn)$ for a proof with m lines and n universally quantified variables [34]. But because this approach constructs a strategy tree for all universal variables at once, the Herbrand functions for individual universal variables have no clear interpretation.

This paper presents a new strategy extraction algorithm for Exp+Res based on Craig interpolation [14]. An *interpolant* between two formulas φ and ψ such that $\varphi \wedge \psi$ is unsatisfiable is a formula (or circuit) I in the shared variables $var(\varphi) \cap var(\psi)$ such that $\varphi \models I$ and $\psi \models \neg I$. Interpolation is an important and well studied concept in logic and automated reasoning [15, 28]. For example, interpolants can be used to over-approximate the set of reachable states in model checking [29]. In proof complexity, showing that an interpolant can be efficiently extracted from a refutation of $\varphi \wedge \psi$ can lead to strong lower bounds [27].

This technique, called feasible interpolation, can also be used to establish lower bounds against QBF proof systems such as Exp+Res when the shared variables are existentially quantified [6]. In this context, it was noted that, for certain formulas, any winning strategy for a single universal variable corresponds to an interpolant. However, extending this idea to multiple universal variables is challenging. In particular, simply computing the interpolants between parts resulting from expansion with u_i and parts resulting from expansion with $\neg u_i$, for each universal variable u_i, does not work for arbitrary Exp+Res proofs (cf. [20], see Section 4.1). This approach only works for *local-first* proofs [20], and bringing a proof into this form generally requires rewriting that can lead to an exponential blowup [1, 22].

This paper presents a different solution that takes the order $u_1, \ldots, u_n$ of universal variables in the quantifier prefix into account. For each universal variable u_i, it computes an interpolant between parts of the complete (Herbrand) expansion that are identified by positive and negative occurrences of u_i along with an assignment σ of the preceding universal variables. For u_1, we simply compute the interpolant between the part resulting from expansion with $\neg u_1$ and the part resulting from expansion with u_1. For u_i with $1 < i \leq n$, we compute an interpolant between the expansion with $\sigma, \neg u_i$ and the expansion with σ, u_i. This not only leads to strategies, it *characterises* them: every universal winning strategy corresponds to such a sequence of interpolants in the complete expansion.

Following this idea, strategy extraction for Exp+Res can by implemented by generalising a standard interpolation system for resolution [21, 27, 33]: interpolants for axioms become functions in universal variables, since their assignment to a part depends on the values of these variables; similarly, whether a variable is shared, or local to a specific part, depends on the assignment of universal variables. The main technical difficulty is showing that an interpolant for a bipartition of the complete expansion can be used as an interpolant between specific parts of the expansion under a partial assignment. This step of the argument is only proved for a specific interpolation system.

The interpolants can be computed in time $O(mn)$ from an Exp+Res proof with m lines and n universal variables, matching the bound of the algorithm that combines partial strategies [34]. We implemented both algorithms within FERPMODELS [19], a certification framework for Exp+Res proofs that uses round-based strategy extraction, and present an experimental comparison of all three algorithms.

The rest of the paper is structured as follows. Section 2 introduces standard concepts and notation. Section 3 offers a brief introduction to interpolation in propositional logic. In Section 4, we establish the link between universal winning strategies and interpolants in the complete expansion, and present the new strategy extraction algorithm for Exp+Res. Section 5 provides experimental results for an implementation of this algorithm. We discuss related work in Section 6 and conclude in Section 7.

2 Preliminaries

An *assignment* of a set V of propositional variables is a function $\sigma : V \to \{0,1\}$. A *partial assignment* of V is an assignment of $U \subseteq V$. Given an assignment $\sigma : V \to \{0,1\}$ and a subset $U \subseteq V$ of its domain, we write $\sigma|_U$ for the restriction of σ to U. We consider (Boolean) circuits and formulas built up from variables, the constants 0 and 1, as well as the connectives $\vee$, $\wedge$, $\neg$. Sometimes, we think of *if-then-else expressions* $\mathsf{ite}(c, A, B)$, which can be expressed as $(c \wedge A) \vee (\neg c \wedge B)$, as atomic gates. We write $var(\varphi)$ for the set of input gates or variables occurring in a circuit or formula φ. If φ is a circuit and $\sigma : V \to \{0,1\}$ an assignment such that $var(\varphi) \subseteq V$, we write $\varphi(\sigma)$ for the output of φ under the assignment. Note that σ may assign variables that are not input gates of φ – these are simply ignored in the evaluation. Given a circuit (or formula) φ and variable assignment $\sigma : V \to \{0,1\}$, we write $\varphi[\sigma]$ for the circuit (or formula) obtained by replacing each input gate (or variable) $v \in var(\varphi) \cap V$ by the constant $\sigma(v)$. A *literal* is a variable v or a negated variable $\neg v$, and a *clause* is a disjunction of literals. A *CNF formula* is a formula is a conjunction of clauses. We think of clauses as sets of literals and formulas as sets of clauses whenever convenient. Similarly, we may identify a variable assignment with a set, sequence, or conjunction of literals. If v is a variable, φ a circuit, and $\tau : V \to \{0,1\}$ an assignment such that $var(\varphi) \subseteq V$, then $v = \varphi(\tau)$ denotes the assignment $\{v \mapsto \varphi(\tau)\}$.

We consider Quantified Boolean Formulas (QBFs) $\Phi = \mathcal{Q}.\varphi$ in prenex conjunctive normal form, where $\mathcal{Q} = Q_1 v_1, \dots, Q_n v_n$ is a sequence of *quantifiers* $Q_i \in \{\forall, \exists\}$ and pairwise distinct variables v_i, called the *(quantifier) prefix* of Φ, and φ is a CNF formula, called the *matrix* of Φ. We assume that the set $var(\varphi)$ of variables in the matrix is a subset of the variables $\{v_1, \dots, v_n\}$ in the prefix. The prefix induces a linear ordering $<_\mathcal{Q}$ on its variables $\{v_1, \dots, v_n\}$ where $v_i <_\mathcal{Q} v_j$ if $i < j$. We omit the prefix $\mathcal{Q}$ when it is understood. Given a partial assignment τ of the variables of Φ, we write $\Phi[\tau] = \mathcal{Q}'.\varphi[\tau]$, where $\mathcal{Q}'$ is obtained from $\mathcal{Q}$ by omitting variables assigned by τ and their associated quantifiers. We write $U_\Phi = \{v_i \mid Q_i = \forall\}$ for the set of *universal* variables of Φ, and $E_\Phi = \{v_i \mid Q_i = \exists\}$ for the set of *existential* variables, dropping the subscript if the QBF Φ is understood. The set of variables preceding variable v_i in the prefix is denoted $D(v_i) = \{v_1, \dots, v_{i-1}\}$. Given a sequence $u_1, \dots, u_k$ of universal variables, we may write $D_i = D(u_i)$. In such cases, we define $D_0 = \emptyset$. Let $\psi = (\psi_1, \dots, \psi_k)$ be a sequence of circuits, one for each universal variable u_i, such that $var(\psi_i) \subseteq E \cup U$. We say that assignments σ of the universal variables and τ of the existential variables are *consistent* with ψ if $\sigma(u_i) = \psi_i(\sigma \cup \tau)$, for each $1 \leq i \leq k$. The sequence ψ is a *universal winning strategy* if $var(\psi) \subseteq D_i$ for each $1 \leq i \leq k$, and $\varphi(\sigma \cup \tau) = 0$ for any assignments σ of the universal variables and τ of the existential variables that are consistent with ψ. The QBF Φ is *false* if there is a universal winning strategy, and *true* otherwise.

2.1 QBF Expansion Proofs

QBF evaluation can be reduced to propositional satisfiability by repeatedly applying Shannon expansion to get rid of universally quantified variables. The resulting propositional formula, called the *complete (Herbrand) expansion*, is satisfiable if, and only if, the QBF is true. The complete expansion can be obtained as a conjunction, taken over all assignments of universal variables, of copies of the matrix instantiated with these assignments. Formally, let $\Phi = \mathcal{Q}.\varphi$ be a QBF, let $C \in \varphi$ be a clause, and let $\sigma : U \to \{0,1\}$ an assignment that does not satisfy C. We write $C^{[\sigma]} = \{\ell^{[\sigma]} \mid \ell \in C, var(\ell) \in E\}$ for the annotated clause obtained by instantiating clause C with the assignment σ, where $\ell^{[\sigma]} = \ell^{\sigma|_{D(var(\ell))}}$ is ℓ annotated with the restriction of σ to universal variables preceding $var(\ell)$ in the prefix. Otherwise, if σ satisfies C, then $C^{[\sigma]} = \top$.

$$\frac{}{C^{[\sigma]}}\ \text{(Axiom)} \qquad\qquad \frac{C_1 \vee e^{\tau} \qquad \neg e^{\tau} \vee C_2}{C_1 \vee C_2}\ \text{(Resolution)}$$

In the axiom rule (left), $C \in \varphi$ is a clause and σ an assignment of universal variables not satisfying C. In the resolution rule (right), both C_1 and C_2 are annotated clauses and e^{τ} is an annotated variable.

Figure 1 The proof rules of Exp+Res for a QBF with matrix φ.

The complete expansion is defined as

$$exp(\Phi) = \bigwedge_{\sigma : U \to \{0,1\}} \bigwedge_{C \in \varphi} C^{[\sigma]}.$$

The complete expansion is satisfiable if, and only if, the QBF Φ is true. Given a partial assignment μ of universal variables, we write $\varphi^{\sigma} = \{C^{[\theta]} \in exp(\Phi) \mid \sigma \subseteq \theta\}$ for the subset of clauses in the complete expansion whose annotation is compatible with σ. Given an assignment τ of existential variables and σ of universal variables, let $\tau^{[\sigma]} = \bigwedge_{\ell \in \tau} \ell^{[\sigma]}$. For a partial assignment α, the expansion of the simplified QBF $\Phi[\alpha]$ essentially corresponds to a subset of the complete expansion of $exp(\Phi)$ with a particular annotation. This is stated formally in the following lemma (the proof is given in Appendix A).

▶ **Lemma 1.** *Let $\Phi = Q_1 v_1 \ldots Q_n v_n . \varphi$ be a QBF, and let $\alpha : \{v_1, \ldots, v_i\} \to \{0,1\}$ be a partial assignment of its variables. Then $exp(\Phi[\alpha])$ and $\varphi^{\sigma} \wedge \tau^{[\sigma]}$ are equisatisfiable, where $\sigma = \alpha|_U$ and $\tau = \alpha|_E$.*

The proof system Exp+Res formally captures resolution refutations from a subset of clauses in the complete expansion [24]. Its proof rules are shown in Figure 1. An Exp+Res *proof* (or *refutation*) of a QBF Φ is a sequence of clauses ending with the empty clause $\bot$ such that each clause is either an axiom or derived by resolution from clauses appearing earlier in the sequence.

3 Interpolation in Propositional Logic

The Craig interpolation theorem states that if $\Phi \models \Psi$ holds for first-order sentences Φ and Ψ, then there exists a first order sentence I, called an *interpolant*, such that $\Phi \models I$, $I \models \Psi$, with the non-logical symbols in I shared by Φ and Ψ [14]. Craig interpolation is an important concept in logic and automated reasoning [28]. Given a propositional formula φ and a clause C, we write $C|_{\varphi}$ for the restriction of C to variables occurring in φ. In the remainder of this paper, we will adopt the following definition of an interpolant, commonly used in model checking and verification [29] and sometimes referred to as a *reverse interpolant* [26].

▶ **Definition 2** (Partial Interpolant). *Let φ and ψ be formulas and C a clause such that $\varphi \wedge \psi \models C$. A* partial interpolant between φ and ψ for C *is a Boolean circuit I such that $var(I) \subseteq var(\varphi) \cap var(\psi)$, $\varphi \models C|_{\varphi} \vee I$, and $\psi \models C|_{\psi} \vee \neg I$. If $C = \emptyset$ then I is called an* interpolant between φ and ψ.

Equivalently, an interpolant between φ and ψ is a circuit that decides which of φ and ψ is unsatisfiable given an assignment of the shared variables.

▶ **Proposition 3.** *Let φ and ψ be formulas such that $\varphi \wedge \psi$ is unsatisfiable. A circuit I with $var(I) \subseteq var(\varphi) \cap var(\psi)$ is an interpolant between φ and ψ if, and only if, $\varphi \wedge \tau$ is unsatisfiable whenever $I(\tau) = 0$, and $\psi \wedge \tau$ is unsatisfiable whenever $I(\tau) = 1$, for any assignment $\tau : var(\varphi) \cap var(\psi) \to \{0,1\}$.*

$$\frac{}{C\ [0]}\ C \in \varphi \qquad\qquad \frac{}{C\ [1]}\ C \in \psi$$

$$\frac{C_1 \vee x\ [I_1] \qquad \neg x \vee C_2\ [I_2]}{C_1 \vee C_2\ [I_1 \vee I_2]}\ x \in \mathit{var}(\varphi) \setminus \mathit{var}(\psi)$$

$$\frac{C_1 \vee x\ [I_1] \qquad \neg x \vee C_2\ [I_2]}{C_1 \vee C_2\ [I_1 \wedge I_2]}\ x \in \mathit{var}(\psi) \setminus \mathit{var}(\varphi)$$

$$\frac{C_1 \vee x\ [I_1] \qquad \neg x \vee C_2\ [I_2]}{C_1 \vee C_2\ [\mathsf{ite}(\neg x, I_1, I_2)]}\ x \in \mathit{var}(\psi) \cap \mathit{var}(\varphi)$$

Figure 2 Symmetric interpolation system for resolution proofs [21, 27, 33].

Proof. Let I be an interpolant between φ and ψ. If $I(\tau) = 0$, since $\varphi \models I$ the formula $\varphi \wedge \tau$ must be unsatisfiable. Otherwise, if $I(\tau) = 1$, since $\psi \models \neg I$, the formula $\psi \wedge \tau$ must be unsatisfiable. The proves the "only if" direction.

For the converse, let I be a circuit defined on $\mathit{var}(\varphi) \cap \mathit{var}(\psi)$ such that $\varphi \wedge \tau$ is unsatisfiable whenever $I(\tau) = 0$, and $\psi \wedge \tau$ is unsatisfiable whenever $I(\tau) = 1$, for any assignment $\tau : \mathit{var}(\varphi) \cap \mathit{var}(\psi) \to \{0, 1\}$. Consider a satisfying assignment τ of φ. Since $\mathit{var}(I) \subseteq \mathit{var}(\varphi)$, the output $I(\tau)$ of I under τ is defined, and it must be 1, since $\varphi \wedge \tau$ is satisfied by τ. We conclude that $\varphi \models I$. A symmetric argument shows that $\psi \models \neg I$. ◀

An *interpolation system* computes circuits representing partial interpolants for each clause in a proof. For the purposes of this paper, we will use the interpolation system for resolution proofs shown in Figure 2 [21, 27, 33]. This interpolation system assigns 0 to initial clauses in φ, and 1 to initial clauses in ψ. For derived clauses, it distinguishes three cases, depending on whether the pivot variable x is *local* to φ, that is, if it may appear in φ but not in ψ, or local to ψ, or *shared* between φ and ψ. We write $I^C(\varphi, \psi)$ for the circuit computed by the system at a clause C of a resolution refutation, and $I(\varphi, \psi) = I^{\emptyset}(\varphi, \psi)$ for the circuit at the empty clause. This circuit is an interpolant, as stated in the following theorem [21, 27, 33].

▶ **Theorem 4.** *Let φ and ψ be formulas such that $\varphi \wedge \psi$ is unsatisfiable. For any resolution refutation of φ and ψ, the circuit $I(\varphi, \psi)$ is an interpolant between φ and ψ.*

Further, the interpolation system is *symmetric* in the following sense [21].

▶ **Lemma 5.** *Let $I'(\varphi, \psi) = I(\psi, \varphi)$ be the circuit computed by the system in Figure 2 where the roles of φ and ψ are reversed. Then $I(\varphi, \psi) \leftrightarrow \neg I'(\varphi, \psi)$.*

4 Strategy Extraction by Interpolation

It is well known that the interpolant between two jointly unsatisfiable formulas identifies which of these formulas is unsatisfiable given an assignment of their shared variables. In particular, that applies to bipartitions of a QBF's expansion induced by individual universal variables, as stated in the following proposition.

▶ **Proposition 6.** *Let $\Phi = \mathcal{Q}.\varphi$ be a false QBF and let u be one of its universal variables. Then $\varphi^{\neg u} \wedge \varphi^{u}$ is unsatisfiable, and for any assignment $\tau : \mathit{var}(\varphi^{\neg u}) \cap \mathit{var}(\varphi^{u}) \to \{0, 1\}$, the formula $\varphi^{u = I(\tau)} \wedge \tau$ is unsatisfiable, where I is an interpolant between $\varphi^{\neg u}$ and φ^{u}.*

$$\dfrac{\dfrac{\dfrac{C_1^{[u_1,\neg u_2]}}{(a)\ [\top,\bot]} \qquad \dfrac{C_3^{[u_1,u_2]}}{(\neg a \lor l^{u_1,u_2})\ [\top,\top]}}{(l^{u_1,u_2})\ [\top,a]} \qquad \dfrac{\dfrac{C_2^{[\neg u_1,u_2]}}{(b)\ [\bot,\top]} \qquad \dfrac{C_4^{[u_1,u_2]}}{(\neg b \lor \neg l^{u_1,u_2})\ [\top,\top]}}{(\neg l^{u_1,u_2})\ [b,\top]}}{\bot\ [b,a]}$$

Figure 3 Exp+Res refutation of the QBF Ψ in the running example. Each clause C is annotated with partial interpolants $[I^C(\psi^{\neg u_1}, \psi^{u_1}), I^C(\psi^{\neg u_2}, \psi^{u_2})]$.

Proof. The formula $\varphi^{\neg u} \land \varphi^{u}$ corresponds to the complete expansion, so it must be unsatisfiable because Φ is false. By Proposition 3, for any assignment $\tau : \mathit{var}(\varphi^{\neg u}) \cap \mathit{var}(\varphi^{u}) \to \{0,1\}$, if $I(\tau) = 0$, then $\varphi^{\neg u} \land \tau$ is unsatisfiable, and if $I(\tau) = 1$, then $\varphi^{u} \land \tau$ is unsatisfiable. ◀

For the first universal variable u in the quantifier prefix, the variables shared between $\varphi^{\neg u}$ and φ^{u} are existential variables preceding u, every interpolant is a function in a winning strategy, and vice versa [6]. However, generalising this correspondence between strategies and interpolants to formulas with multiple universal variables is non-trivial. We first consider a natural but unsuccessful approach in Section 4.1 before presenting a solution in Section 4.2.

4.1 A Naive Approach

An initially plausible idea for obtaining a winning strategy is to separately compute the interpolant between $\varphi^{\neg u_i}$ and φ^{u_i} for each universal variable u_i. Unfortunately, that does not work in general because functions obtained in this way lack coordination, as illustrated by the following example [20].

▶ Example. Consider the QBF $\Psi = \exists a \exists b \forall u_1, \forall u_2 \exists l . \psi$, where

$$\psi = \underbrace{(a \lor \neg u_1 \lor u_2)}_{C_1} \land \underbrace{(b \lor u_1 \lor \neg u_2)}_{C_2} \land \underbrace{(\neg a \lor \neg u_1 \lor \neg u_2 \lor l)}_{C_3} \land \underbrace{(\neg b \lor \neg u_1 \lor \neg u_2 \lor \neg l)}_{C_4} .$$

The QBF Ψ is false, as witnessed by the Exp+Res refutation shown in Figure 3. Taking this proof as a resolution refutation of the expansion $\mathit{exp}(\Psi)$, we can apply the interpolation system in Figure 2 to compute the interpolants $I(\psi^{\neg u_1}, \psi^{u_1}) = b$ and $I(\psi^{\neg u_2}, \psi^{u_2}) = a$. However, (b, a) is not a universal winning strategy, since the satisfying assignment $\neg a, \neg b, \neg u_1, \neg u_2, l$ of ψ is consistent with this strategy.

This issue can be circumvented by working with resolution refutations of the expansion that are *local-first*, where resolution on shared pivot variables may occur only after local variables have been removed by resolution [20]. However, imposing this kind of proof structure may require rewriting and can lead to an exponential increase in proof size [1, 22].

4.2 Coordinated Interpolants are Strategies

To achieve coordination between interpolants, we will take the ordering of universal variables in the quantifier prefix into account. To simplify notation, for the rest of this section, let $\Phi = \mathcal{Q}.\varphi$ be an arbitrary but fixed, false QBF with n universal variables $u_1, \ldots, u_n$, in their left-to-right order in the quantifier prefix.

For the first variable u_1, we compute an interpolant I_1 between $\varphi^{\neg u_1}$ and φ^{u_1}, as suggested above. Given an assignment $\tau : \mathit{var}(\varphi^{\neg u_1}) \cap \mathit{var}(\varphi^{u_1}) \to \{0,1\}$ of the shared variables, the interpolant computes an assignment $I_1(\tau)$ such that $\varphi^{u_1 = I_1(\tau)} \land \tau$ is unsatisfiable.

We generalise this to an inductive invariant for $1 < i \leq n$ by requiring that the partial assignment $\sigma : \{u_1, \ldots, u_{i-1}\} \to \{0,1\}$ of universal variables identifies a part φ^σ of the complete expansion that is unsatisfiable under the partial assignment $\tau : D_{i-1} \cap E \to \{0,1\}$ of existential variables. Technically, φ^σ speaks about annotated existential variables, rather than the original existential variables. Since their annotations are all consistent with σ (by definition of φ^σ), this is not really an issue, but just to be formally precise, we add the annotation to the assignment τ and require that $\varphi^\sigma \wedge \tau^{[\sigma]}$ is unsatisfiable.

To obtain a strategy function for u_i, we now compute an interpolant *within* φ^σ. Since φ^σ is the union of $\varphi^{\sigma,\neg u_i}$ and φ^{σ,u_i}, and $\varphi^\sigma \wedge \tau^{[\sigma]}$ is unsatisfiable, there is an interpolant between $\varphi^{\sigma,\neg u_i} \wedge \tau^{[\sigma]}$ and $\varphi^{\sigma,u_i} \wedge \tau^{[\sigma]}$. We will call such an interpolant a *σ,τ-interpolant.*

▶ **Definition 7** (σ,τ-Interpolant). *Let $1 \leq i \leq n$, let $\sigma : \{u_1, \ldots, u_{i-1}\} \to \{0,1\}$ be a partial assignment of universal variables and $\tau : D_{i-1} \cap E \to \{0,1\}$ a partial assignment of existential variables such that $\varphi^\sigma \wedge \tau^{[\sigma]}$ is unsatisfiable. A σ,τ-*interpolant *is an interpolant between $\varphi^{\sigma,\neg u_i} \wedge \tau^{[\sigma]}$ and $\varphi^{\sigma,u_i} \wedge \tau^{[\sigma]}$.*

Given an assignment of their shared variables, a σ,τ-interpolant will determine which of the two formulas $\varphi^{\sigma,\neg u_i} \wedge \tau^{[\sigma]}$ and $\varphi^{\sigma,u_i} \wedge \tau^{[\sigma]}$ is unsatisfiable, and maintain our invariant. But it will only do that for the specific assignments σ and τ. To obtain a strategy, we need functions that compute σ,τ-interpolants given assignments σ,τ. Just like the formula φ^σ, a σ,τ-interpolant is defined on annotated existential variables. However, since the interpolant can only use variables shared between $\varphi^{\sigma,\neg u_i}$ and φ^{σ,u_i}, for each existential variable e, the only annotated variable that can appear in the interpolant is $e^{[\sigma]}$. That allows us to use circuits defined on the *original* variables to compute σ,τ-interpolants by renaming (annotating) the input variables. Extending our notation $C^{[\sigma]}$ for annotated clauses, we write $I^{[\sigma]}$ for the circuit obtained from I by replacing universal input gates u in the domain of σ by the constant gate $\sigma(u)$, and replacing each existential input gate e by the annotated gate $e^{[\sigma]}$. Following Hofferek et al. [20], we call a sequence of circuits computing σ,τ-interpolants an *n-interpolant.*

▶ **Definition 8** (n-Interpolant). *An n-*interpolant *is a sequence $\mathbf{I} = (I_1, \ldots, I_n)$ of circuits with the following properties:*

(a) *Each I_i is defined on variables D_i, for $1 \leq i \leq n$.*

(b) *For any pair of assignments $\sigma : U \to \{0,1\}$ and $\tau : E \to \{0,1\}$ consistent with $\mathbf{I}$, the circuit $I_i^{[\sigma_{i-1}]}$ is a σ_{i-1},τ_{i-1}-interpolant whenever $\varphi^{[\sigma_{i-1}]} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ is unsatisfiable, for each $1 \leq i \leq n$.*

Here, $\sigma_i = \sigma|_{\{u_1,\ldots,u_i\}}$ and $\tau_i = \tau|_{D_i}$.

We first prove that an n-interpolant of a false QBF is a universal winning strategy.

▶ **Proposition 9.** *An n-interpolant is a universal winning strategy.*

Proof. Let $\mathbf{I} = (I_1, \ldots, I_n)$ be an n-interpolant, and let $\sigma : U \to \{0,1\}$ and $\tau : E \to \{0,1\}$ be assignments consistent with $\mathbf{I}$, formally $I_i(\sigma \cup \tau) = \sigma(u_i)$ for $1 \leq i \leq n$. Further, for $1 \leq i \leq n$, let $\sigma_i = \sigma|_{\{u_1,\ldots,u_i\}}$ and $\tau_i = \tau|_{D_i}$ denote restrictions of these assignments as in Definition 8.

We show that $\varphi^{\sigma_i} \wedge \tau_i^{[\sigma_i]}$ is unsatisfiable for $0 \leq i \leq n$. For $\sigma_n = \sigma$, since σ is a complete assignment of universal variables, the annotated formula φ^σ is syntactically equivalent to the restriction $\varphi[\sigma]$ when annotations are dropped. So if $\varphi^{[\sigma]} \wedge \tau^{[\sigma]}$ is unsatisfiable, the matrix φ must be falsified by $\sigma \cup \tau$. Since τ was chosen arbitrarily, this would prove that the n-interpolant is a universal winning strategy.

We proceed by induction on i. For $i = 0$, the assignments σ_0 and τ_0 are empty, and $\varphi^{\sigma_0} \wedge \tau_0^{[\sigma_0]}$ coincides with the Herbrand expansion $\mathit{exp}(\Phi)$, which is unsatisfiable because the QBF Φ is assumed to be false. Suppose the statement holds up to $i - 1 < n$. By definition, $I_i^{[\sigma_{i-1}]}$ is a σ_{i-1}, τ_{i-1}-interpolant, and since $\varphi^{\sigma_{i-1}} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ is unsatisfiable by induction hypothesis, $I_i^{[\sigma_{i-1}]}$ is an interpolant between $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$. Since I_i is defined on variables D_i, the assignment $\tau_i^{[\sigma_{i-1}]}$ assigns all variables of $I_i^{[\sigma_{i-1}]}$, and $I_i^{[\sigma_{i-1}]}(\tau_i^{[\sigma_{i-1}]}) = I_i(\sigma_{i-1} \cup \tau_i) = \sigma(u_i)$. By Proposition 3, $\varphi^{\sigma_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]} \wedge \tau_i^{[\sigma_i]}$ is unsatisfiable, and since $\tau_{i-1}^{[\sigma_{i-1}]} \subseteq \tau_i^{[\sigma_i]}$, that is the same as saying that $\varphi^{\sigma_i} \wedge \tau_i^{[\sigma_i]}$ is unsatisfiable. ◀

The converse is true as well: every universal winning strategy is an n-interpolant. In combination, we get the following result.

▶ **Theorem 10.** *A sequence of circuits is a universal winning strategy if, and only if, it is an n-interpolant.*

Proof. The "if" direction follows from Proposition 9. For the "only if" direction, consider a universal winning strategy $\mathbf{S} = (S_1, \ldots S_n)$, and let $\sigma : U \to \{0,1\}$ and $\tau : E \to \{0,1\}$ be assignments consistent with $\mathbf{S}$. As before, let $\sigma_i = \sigma|_{\{u_1,\ldots,u_i\}}$ and $\tau_i = \tau|_{D_i}$ for $0 \le i \le n$. Any strategy trivially satisfies property (a). To prove that $\mathbf{S}$ is an n-interpolant, we additionally have to show (b) that for each $1 \le i \le n$, the circuit $S_i^{[\sigma_{i-1}]}$ is an σ_{i-1}, τ_{i-1}-interpolant – that is, an interpolant between $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$. Suppose $\varphi^{\sigma_{i-1}} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ is unsatisfiable. We will prove that $S_i^{[\sigma_{i-1}]}$ correctly identifies an unsatisfiable formula among $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$, given an assignment of the shared variables. These are variables in $\tau_{i-1}^{[\sigma_{i-1}]}$, as well as variables shared between $\varphi^{\sigma_{i-1},\neg u_i}$ and $\varphi^{\sigma_{i-1},u_i}$, which are variables $e^{[\sigma_{i-1}]}$ for some $e \in D_i \cap E$. Since τ_{i-1} is an assignment of $D_{i-1} \cap E$, and $D_{i-1} \subseteq D_i$, every shared variable is an annotated variable $e^{[\sigma_{i-1}]}$ for $e \in D_i \cap E$. Now consider an arbitrary assignment $\nu : D_i \cap E \to \{0,1\}$, and its annotated version $\nu^{[\sigma_{i-1}]}$. If $\nu^{[\sigma_{i-1}]}$ is inconsistent with $\tau_{i-1}^{[\sigma_{i-1}]}$, then both formulas $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ are unsatisfiable under this assignment, so we can assume that $\nu^{[\sigma_{i-1}]}$ extends $\tau_{i-1}^{[\sigma_{i-1}]}$. It follows that the responses of strategy $\mathbf{S}$ must coincide with assignment σ for universal variables preceding u_i, formally $S_j(\nu) = \sigma(u_j)$ for each $1 \le j < i$. Let σ' denote the assignment σ_{i-1} extended by assigning $\sigma'(u_i) = S_i(\nu)$. The assignments σ' and ν are consistent with $\mathbf{S}$, so like above, we can conclude that $\varphi^{\sigma'} \wedge \nu^{[\sigma']}$ is unsatisfiable. Assume first that $\sigma'(u_i) = 0$. Since ν only assigns variables to the left of u_i, variable u_i does not show up in annotations and $\nu^{[\sigma']} = \nu^{[\sigma_{i-1}]}$. Further, recall that $\nu^{[\sigma_{i-1}]}$ extends $\tau_{i-1}^{[\sigma_{i-1}]}$. Thus $\varphi^{\sigma'} \wedge \nu^{[\sigma']} = \varphi^{\sigma,\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]} \wedge \nu^{[\sigma']} = \varphi^{\sigma,\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]} \wedge \nu^{[\sigma_{i-1}]}$ is unsatisfiable. Similarly, if $\sigma'(u_i) = 1$, then $\varphi^{\sigma,u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]} \wedge \nu^{[\sigma_{i-1}]}$ is unsatisfiable. By Proposition 3, $S_i^{[\sigma_{i-1}]}$ is an interpolant between $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$, as claimed. ◀

4.3 Computing Coordinated Interpolants from Exp+Res Proofs

Theorem 10 does not refer to a proof system or interpolation algorithm. In this section, we will show that an n-interpolant representing a universal winning strategy can be computed from an Exp+Res refutation in time $O(mn)$, where m is the number of clauses in the refutation.

We use the interpolation system shown above in Figure 2, in combination with a function that assigns clauses and variables to parts depending on a partial assignment of universal variables. Let u_i be a universal variable and $\sigma : \{u_1, \ldots, u_{i-1}\} \to \{0,1\}$ an assignment of universal variables that precede it in the prefix, and let $p \in \{0,1\}$ be a truth value for variable u_i. We will compute an interpolant between $\varphi^{\sigma,u_i=p}$ and its complement $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$ in the complete expansion. Such an interpolant must exist because the complete expansion is unsatisfiable. Each clause C is assigned a partial interpolant I^C as follows:

- If C is an initial clause, then $I^C = 0$ if $C \in \varphi^{\sigma,u_i=p}$ and $I^C = 1$ otherwise.[1]
- If C is derived by resolution from clauses $C_1 \vee e^\mu$ and $\neg e^\mu \vee C_2$ with partial interpolants I^1 and I^2, we distinguish two cases:
  - **(I)** If $e < u_i$, then u_i does not appear in the annotation μ, and there are two options:
    - **(a)** If σ is consistent with μ, then e^μ is a shared variable, and $I^C = \mathsf{ite}(\neg e^\mu, I^1, I^2)$.
    - **(b)** Otherwise, if μ is not consistent with σ, then e^μ is local to $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$ and $I^C = I^1 \wedge I^2$.
  - **(II)** If $e > u_i$, then the annotation μ contains a u_i-literal, and again there are two cases:
    - **(a)** If μ is consistent with $\sigma \wedge (u_i = p)$, then e^μ is local to $\varphi^{\sigma,u_i=p}$, and $I^C = I^1 \vee I^2$.
    - **(b)** Otherwise, if μ is inconsistent with $\sigma \wedge (u_i = p)$, then e^μ is local to $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$, and $I^C = I^1 \wedge I^2$.

Construction of Interpolant Circuits

The above definition lets us compute an interpolant between $\varphi^{\sigma,u_i=p}$ and $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$ for a fixed assignment σ of universal variables. By instead considering the universal variables $u_1, \dots, u_{i-1}$ as inputs, we can construct circuits I_i^C that take this assignment σ as an input and compute partial interpolants between $\varphi^{\sigma,u_i=p}$ and $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$ for each clause C.

For a given annotation μ and index i with $0 \leq i \leq n$, let H_μ^i denote a circuit that compares μ and with its input $\sigma : \{u_1, \dots, u_i\} \to \{0,1\}$ and outputs 1 if $\mu(u_j) = \sigma(u_j)$ for all $1 \leq j \leq n$:

$$H_\mu^i := \bigwedge_{j=1}^{i} \mu(u_j) \leftrightarrow u_j$$

With this, we define circuits I_i^C for each index $1 \leq i \leq n$ and clause C in the refutation, where $p \in \{0,1\}$ is a constant as above:

- If $C^{[\mu]}$ is an axiom, then $I_i^C := \neg\left(H_\mu^{i-1} \wedge \mu(u_i) \leftrightarrow p\right)$.
- Otherwise, if C is derived by resolution from clauses $C_1 \vee e^\mu$ and $\neg e^\mu \vee C_2$ with partial interpolants I_i^1 and I_i^2, then we let I_i^C be one of the following two circuits, depending on the order of e and u_i (which is independent of the assignment σ):
  - **(I)** If $e < u_i$, let k be the maximum index such that $u_k < e$, and
  $$I_i^C := \mathsf{ite}(H_\mu^k, \mathsf{ite}(\neg e, I_i^1, I_i^2), I_i^1 \wedge I_i^2).$$
  - **(II)** Otherwise, if $e > u_i$, let $G := H_\mu^{i-1} \wedge \mu(u_i) \leftrightarrow p$, and
  $$I_i^C := \mathsf{ite}(G, I_i^1 \vee I_i^2, I_i^1 \wedge I_i^2).$$

We write $I_i := I_i^\emptyset$ for the circuits constructed at the empty clause.

[1] Here, we assume that the complete assignment of universal variables used in the axiom rule is given. In the implementation, where this full assignment is not part of the proof, we can assume that all universal variables missing from annotations were assigned 0. A minor optimisation is to leave their assignments open, and only fix them once we see a resolution step where the other premise has a partial interpolant for such a variable.

$$\dfrac{\dfrac{\dfrac{C_1^{[u_1,\neg u_2]}}{(a)\ [\top,\neg u_1]} \qquad \dfrac{C_3^{[u_1,u_2]}}{(\neg a \vee l^{u_1,u_2})\ [\top,\top]}}{(l^{u_1,u_2})\ [\top, a \vee \neg u_1]} \qquad \dfrac{\dfrac{C_2^{[\neg u_1,u_2]}}{(b)\ [\bot,\top]} \qquad \dfrac{C_4^{[u_1,u_2]}}{(\neg b \vee \neg l^{u_1,u_2})\ [\top,\top]}}{(\neg l^{u_1,u_2})\ [b,\top]}}{\bot\ [b, a \vee \neg u_1]}$$

Figure 4 Exp+Res refutation of Figure 3, but each clause C is annotated with coordinated interpolants $[I_1^C, I_2^C]$.

▶ Example (continued). Figure 4 shows the circuits I_1^C, I_2^C for each clause C of the Exp+Res refutation from Figure 3. The circuits I_1^C are identical to the partial interpolants $I^C(\psi^{\neg u_1}, \psi^{u_1})$ computed before, but the circuits I_2^C compute partial interpolants between $\psi^{\sigma,\neg u_2}$ and $exp(\Psi) \setminus \psi^{\sigma,\neg u_2}$, where σ is an unknown assignment of u_1. For instance, whether the axiom $C_1^{[u_1,\neg u_2]}$ is in $\psi^{\sigma,\neg u_2}$ or not depends on the assignment $\sigma(u_1)$: if $\sigma(u_1) = 1$, then clause $C_1^{[u_1,\neg u_2]}$ is in $\psi^{\sigma,\neg u_2}$ and should receive the label $\bot$; otherwise, the label should be $\top$. Accordingly, its partial interpolant is simply $\neg u_1$. On the other hand, clause $C_3^{[u_1,u_2]}$ cannot be in $\psi^{\sigma,\neg u_2}$ simply because it was instantiated with literal u_2, so we can immediately set its partial interpolant to $\top$. The same is true of both axioms on the right side of the proof tree. Similarly, the final resolution step on pivot l^{u_1,u_2} is local to $exp(\Psi) \setminus \psi^{\sigma,\neg u_2}$, and so the partial interpolant for the resolvent is computed as $(a \vee \neg u_1) \wedge \top \equiv a \vee \neg u_1$.

It is readily verified that the interpolants $(b, a \vee \neg u_1)$ are a universal winning strategy. In particular, for the existential assignment $\neg a \wedge \neg b$, which led to a counterexample for the naive approach, it computes the assignment $\neg u_1 \wedge u_2$, and the joint assignment falsifies clause C_2.

▶ **Lemma 11.** *Let $p \in \{0,1\}$ be a constant. For every assignment $\sigma : \{u_1, \ldots, u_{i-1}\} \to \{0,1\}$ and $1 \le i \le n$, the circuit $I_i^{[\sigma]}$ is an interpolant between $\varphi^{\sigma,u_i=p}$ and $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$.*

Proof. For each circuit I_i^C, applying the assignment σ yields a circuit $I_i^{C[\sigma]}$ that is equivalent to the circuit I^C computed by the symmetric interpolation system for clause C and assignment σ, and this circuit I^C is a partial interpolant between $\varphi^{\sigma,u_i=p}$ and $exp(\Phi) \setminus \varphi^{\sigma,u_i=p}$ for C. ◀

By sharing subcircuits, a circuit with one output for each I_i can be computed from an Exp+Res refutation in a single pass.

▶ **Proposition 12.** *Let $p \in \{0,1\}$ be a constant. A circuit with n outputs computing I_i for each $1 \le i \le n$ can be constructed in time $O(mn)$.*

Proof. For each annotation μ, a circuit computing H_μ^i for each $1 \le i \le n$ can be constructed in time $O(n)$ by using the fact that $H_\mu^{i+1} \leftrightarrow H_\mu^i \wedge (\mu(u_{i+1}) \leftrightarrow u_{i+1})$ for $0 \le i < n$. For each clause C, the circuit I_i^C can be constructed in constant time from H_μ^j with $1 \le j \le i$ and circuits I_i^B for clauses B preceding C in the refutation. So computing a circuit with outputs representing I_i^C for a clause C takes time $O(n)$, and there are m clauses in the refutation, so it takes time $O(mn)$ to construct a circuit with outputs representing I_i for $1 \le i \le n$. ◀

Unless otherwise stated, we we let $p = 0$, and compute interpolants I_i between $\varphi^{\sigma,\neg u_i}$ and $exp(\Phi) \setminus \varphi^{\sigma,\neg u_i}$. It remains to show that these can be used as interpolants between $\varphi^{\sigma,\neg u_i} \wedge \tau^{[\sigma]}$ and $\varphi^{\sigma,u_i} \wedge \tau^{[\sigma]}$. That is not trivial, because an interpolant between $\varphi^{\sigma,\neg u_i}$ and $exp(\Phi) \setminus \varphi^{\sigma,\neg u_i}$ may output 1 if $exp(\Phi) \setminus \varphi^{\sigma,\neg u_i}$ is unsatisfiable even when φ^{σ,u_i} is satisfiable. However, we can rule out this case for interpolants computed by the symmetric interpolation system and prove the following result.

▶ **Proposition 13.** *The sequence* $\mathbf{I} = (I_1, \dots, I_n)$ *is an* n*-interpolant.*

Proof. Each circuit I_i takes variables from D_i as inputs, thus satisfying Part (a) of Definition 8. For Part (b), let $\tau : E \to \{0,1\}$ be an assignment of existential variables and $\sigma : \{u_1, \dots, u_n\} \to \{0,1\}$ an assignment of universal variables consistent with $\mathbf{I}$. We have to show that $I_i^{[\sigma_{i-1}]}$ is a σ_{i-1}, τ_{i-1}-interpolant for each $1 \leq i \leq n$, where $\sigma_i = \sigma|_{\{u_1,\dots,u_i\}}$ and $\tau_i = \tau|_{D_i}$. That is, we must demonstrate that $I_i^{[\sigma_{i-1}]}$ is an interpolant between $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ whenever $\varphi^{\sigma_{i-1}} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ is unsatisfiable.

Let $\mathbf{J} = (J_1, \dots, J_n)$ be the sequence of "dual" interpolants for $p = 1$ between φ^{σ,u_i} and $exp(\Phi) \setminus \varphi^{\sigma,u_i}$. By Lemma 11 and Proposition 3 in combination with unsatisfiability of $exp(\Phi)$, if $I_i^{[\sigma_{i-1}]}$ outputs 0, then $\varphi^{\sigma,\neg u_i} \wedge \tau_i^{[\sigma_i]}$ is unsatisfiable, and if $J_i^{[\sigma_{i-1}]}$ outputs 0, then $\varphi^{\sigma,u_i} \wedge \tau_i^{[\sigma_i]}$ is unsatisfiable. By induction on i, we will show that whenever the circuit $I_i^{[\sigma_{i-1}]}$ outputs 1, circuit $J_i^{[\sigma_{i-1}]}$ outputs 0. Proposition 3 then tells us that $I_i^{[\sigma_{i-1}]}$ is an interpolant between $\varphi^{\sigma_{i-1},\neg u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$ and $\varphi^{\sigma_{i-1},u_i} \wedge \tau_{i-1}^{[\sigma_{i-1}]}$, as required.

For $i = 1$, this follows from the symmetry of the interpolation system as stated in Lemma 5 and the fact that $exp(\Phi) \setminus \varphi^{\neg u_i} = \varphi^{u_i}$. Let $1 < i \leq n$ and assume without loss of generality that $\sigma(u_{i-1}) = I_{i-1}(\sigma \cup \tau) = 0$ (if $\sigma(u_{i-1}) = 1$, we simply apply the induction hypothesis to obtain $J_{i-1}(\sigma \cup \tau) = 0$ and work with J_{i-1} instead). We now claim that $I_i^{[\sigma_{i-1}]}(\tau_i^{[\sigma_{i-1}]}) = 1$ and $J_i^{[\sigma_{i-1}]}(\tau_i^{[\sigma_{i-1}]}) = 1$ imply $I_{i-1}^{[\sigma_{i-2}]}(\tau_{i-1}^{[\sigma_{i-2}]}) = 1$. Since $I_{i-1}(\sigma \cup \tau) = I_{i-1}^{[\sigma_{i-2}]}(\tau_{i-1}^{[\sigma_{i-2}]}) = 0$, it would follow that whenever I_i outputs 1, J_i must output 0.

To prove this claim, we compare the circuits $I_i^{[\sigma_{i-1}]}$, $J_i^{[\sigma_{i-1}]}$, and $I_{i-1}^{[\sigma_{i-2}]}$. Since they all come from the same Exp+Res proof, they share its structure, and there is a one-to-one correspondence between their gates. More specifically, we obtain gates in $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$ from gates in $I_{i-1}^{[\sigma_{i-2}]}$ as follows:

1. 0-gates coming from initial clauses in $\varphi^{\sigma_{i-1},u_i}$ become 1-gates in $I_i^{[\sigma_{i-1}]}$ and remain 0-gates in $J_i^{[\sigma_{i-1}]}$. Symmetrically, 0-gates coming from initial clauses in $\varphi^{\sigma_{i-1},\neg u_i}$ become 1-gates in $J_i^{[\sigma_{i-1}]}$ but remain 0-gates in $I_i^{[\sigma_{i-1}]}$.
2. 1-gates coming from initial clauses in $exp(\Phi) \setminus \varphi^{\sigma_{i-1}}$ remain 1-gates in both $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$, since $exp(\Phi) \setminus \varphi^{\sigma_{i-1}} \subseteq exp(\Phi) \setminus \varphi^{\sigma_{i-1},\ell}$ for $\ell \in \{u_i, \neg u_i\}$.
3. $\vee$-gates from resolution steps with pivots e^{σ_{i-1},u_i} local to $\varphi^{\sigma_{i-1},u_i}$ become $\wedge$-gates in $I_i^{[\sigma_{i-1}]}$, and $\vee$-gates from resolution steps with pivots $e^{\sigma_{i-1},\neg u_i}$ local to $\varphi^{\sigma_{i-1},\neg u_i}$ become $\wedge$-gates in $J_i^{[\sigma_{i-1}]}$.
4. $\vee$-gates from resolution steps with pivots $e^{\sigma_{i-1}}$ shared between $\varphi^{\sigma_{i-1},\neg u_i}$ and $\varphi^{\sigma_{i-1},u_i}$, but local to $\varphi^{\sigma_{i-1}}$, become ite-gates in $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$.
5. $\wedge$-gates coming from resolution steps local to $exp(\Phi) \setminus \varphi^{\sigma_{i-1}}$ remain $\wedge$-gates.
6. ite-gates coming from resolution steps on variables shared between $\varphi^{\sigma_{i-1}}$ and $exp(\Phi)\setminus\varphi^{\sigma_{i-1}}$ remain ite-gates. That is because any such shared variable must be of the form e^{σ_k} for some $k < i-1$, and so it will also be shared between $\varphi^{\sigma_{i-1},\ell}$ and $exp(\Phi) \setminus \varphi^{\sigma_{i-1},\ell}$ for $\ell \in \{u_i, \neg u_i\}$.

We now show, by induction on the position of a clause in the proof, that whenever its corresponding gate outputs 1 under assignment $\tau^{[\sigma_{i-1}]}$ in both $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$, then the gate must output 1 in $I_{i-1}^{[\sigma_{i-2}]}$ as well. We argue separately for each of the above cases:

1. If the clause is an axiom in $\varphi^{\sigma_{i-1}}$, then we get contradicting constant gates in $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$, and the statement holds trivially.
2. For axioms in $exp(\Phi) \setminus \varphi^{\sigma_{i-1}}$, we get 1-gates in all three circuits, so the statement again holds trivially.

3. For a clause derived by resolution on a pivot variable local to $\varphi^{\sigma,\neg u_i}$ or φ^{σ,u_i}, we get an $\vee$-gate in $I_i^{[\sigma_{i-1}]}$ and an $\wedge$-gate in $J_i^{[\sigma_{i-1}]}$, or vice versa. In either case, if both the $\wedge$-gate and the $\vee$-gate output 1, then there has to be an input that is 1 in both $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$. By induction hypothesis, this input must also be 1 in $I_{i-1}^{[\sigma_{i-2}]}$, and because we have an $\vee$-gate for this clause in $I_{i-1}^{[\sigma_{i-2}]}$, its output must be 1 as well.
4. If the pivot is shared between $\varphi^{\sigma_{i-1},\neg u_i}$ and $\varphi^{\sigma_{i-1},u_i}$, but local to $\varphi^{\sigma_{i-1}}$, then we get ite-gates in $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$ that take their values from the same input under assignment $\tau^{[\sigma_{i-1}]}$. If the output of the gate in both circuits is 1, that input must be 1 in both circuits, and thus also in $I_{i-1}^{[\sigma_{i-2}]}$ by induction hypothesis. Since we get an $\vee$-gate for this clause in $I_{i-1}^{[\sigma_{i-2}]}$, its output must be 1.
5. If the pivot is local to $exp(\Phi) \setminus \varphi^{\sigma_{i-1}}$, then we get an $\wedge$-gate in all three circuits, and if the gates in $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$ output 1, we can again apply the induction hypothesis to the inputs to conclude that the gate's output has to be 1 in $I_{i-1}^{[\sigma_{i-2}]}$ as well.
6. Finally, if the pivot is shared between $\varphi^{\sigma_{i-1}}$ and $exp(\Phi) \setminus \varphi^{\sigma_{i-1}}$, we get ite-gates in all circuits, taking their values from the same input under the assignment $\tau^{[\sigma_{i-1}]}$. We can once again apply the induction hypothesis to this input to conclude that the output of the gate in $I_{i-1}^{[\sigma_{i-2}]}$ has to be 1.

This completes the induction argument. In particular, whenever the circuits $I_i^{[\sigma_{i-1}]}$ and $J_i^{[\sigma_{i-1}]}$ both output 1, then $I_{i-1}^{[\sigma_{i-2}]}$ must output 1 as well, proving the claim. ◀

5 Experiments

We implemented the algorithm described in Section 4.2 within FERPModels,[2] a framework for strategy extraction from Exp+Res proofs that supports round-based strategy extraction [19]. For reference, we also implemented strategy extraction based on combination of partial strategies [34]. The modified version of FERPModels is available on GitHub.[3]

5.1 Setup

The pipeline for extracting and validating strategies for a false QBF in FERPModels includes the following steps:

1. Solving the QBF with the expansion solver Ijtihad [9].
2. Using the SAT solver PicoSAT [7] to generate a proof of unsatisfiability of the final expansion in the TraceCheck format [36].
3. Generating and validating a FERP proof, which maps variables in the unsatisfiability proof to annotated variables, and initial clauses to Exp+Res axioms.
4. Extracting the strategy as an AND-Inverter Graph (AIG) from the FERP proof.
5. Validating the strategy by conjoining its CNF encoding with the matrix of the QBF using QBFCert [30], and proving unsatisfiability of the resulting (propositional) formula. Since these SAT calls are frequently a bottleneck, we decided to use CaDiCaL[4] here instead of the default PicoSAT.

[2] `https://github.com/SFMV/ferp-models`
[3] `https://github.com/fslivovsky/ferpmodels`
[4] `https://github.com/arminbiere/cadical`

The only step that varies between different versions used in our experiments is Step 4, so we get an apples-to-apples comparison of strategy extraction algorithms. We refer to the three versions as *interpolant*, *combine*, and *round-based*.

For our experiments, we used a cluster with AMD EPYC 7402 CPUs running 64-bit Linux. We first identified 135 instances from the PCNF track of QBFEval 2020 that could be solved by IJTIHAD within 15 minutes and with a memory limit of 8 GB. Of these, 92 are false and were considered further for strategy extraction. For each of these formulas and each strategy extraction algorithm, we ran the entire pipeline described above (including QBF solving) once, using a time limit of 30 minutes and a memory limit of 32 GB.

5.2 Results

For 4 out of 92 instances, the strategy extraction step was not reached:

- For 3 instances, the proof generation in Step 2 failed. More specifically, for 2 instances, PICOSAT was unable to solve the expansion (again) within the timeout. For 1 instance, checking the UNSAT proof timed out.
- For 1 instance, the FERP trace generation in Step 3 ran out of memory.[5]

The numbers of extracted and verified strategies for the remaining 88 instances are shown in Figure 5.

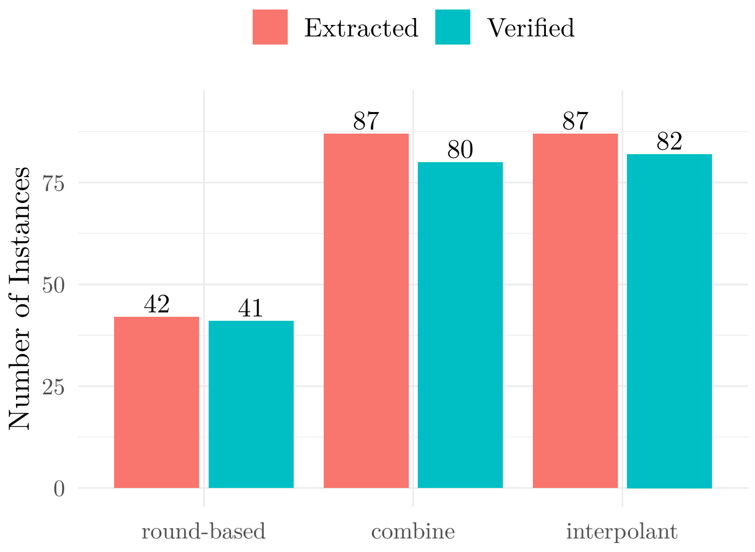


Figure 5 Number of extracted and verified strategies, by algorithm.

Strategies could be extracted by both *interpolant* and *combine* for 87 out of 88 instances (for the remaining instance, Steps 1-3 take about 26 minutes, not leaving enough time for strategy extraction), compared to 42 instances with *round-based*.[6] The numbers for verified strategies follow a similar trend, with *combine* and *interpolant* seeing 80 and 82 verified strategies respectively, compared to 41 for *round-based*. Notably, the instances where

[5] This could perhaps be addressed by switching from a binary resolution proof generated by TRACECHECK to a more succinct proof format, but such optimisations are beyond of the scope of this work.
[6] This number is slightly lower than the one reported in the original paper [19], probably because of a more restrictive memory limit in our experiments (32 instead of 50 GB).

strategies could be extracted by *interpolant* and *combine* are the same, and these include all instances where strategies could be extracted by *round-based*. Similarly, if a strategy for an instance could be verified with *round-based*, it could be verified with *combine*, and every instance verified with *combine* could be verified with *interpolant*.

As in previous experiments on strategy extraction [19, 30, 31], the validation step takes up a significant fraction of the overall running time.

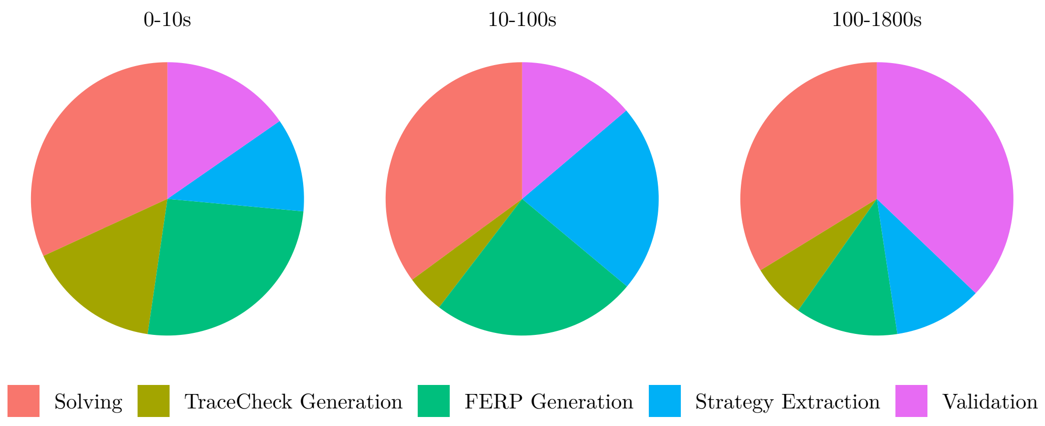


Figure 6 Average fraction of running time spent on each step of the strategy extraction and validation pipeline, grouped by overall running time.

Figure 6 shows that the longer the overall running time, the more time is spent on validation. Figure 7 shows the number of instances for which strategies could be extracted within a given time budget, for each algorithm. The running times for *combine* and *interpolant* are very similar, and both algorithms can extract strategies for all but one instance within 250 seconds.

Figure 8 compares the number of nodes between AIGs for strategies extracted by the three algorithms. As one would expect given the gap in running times, the strategies extracted by *round-based* generally require much bigger AIGs than the strategies extracted by *interpolant*. On average (geometric mean), the AIGs for *round-based* are about 30 times larger than for *interpolant*. The biggest difference we saw was an instance where *interpolant* (and *combine*) extract a strategy with 160 nodes, while the AIG for *round-based* required more than 100 million nodes. However, there was also an instance where the *round-based* strategy required only about 600 nodes, compared to 20000 for *interpolant*. As the figure shows, the AIG sizes are much more similar between *interpolant* and *combine*. On average, the strategies for *interpolant* are larger by a factor of 1.6. There is an instance where the AIG for *combine* has only about 2000 nodes, while the AIG for *interpolant* has 45000 nodes. Conversely, there is a QBF where the strategy for *interpolant* requires 26000 nodes, while the strategy for *combine* requires 48000 nodes. Finally, the strategy size achieved with *combine* was never larger than the strategy size with *round-based*.

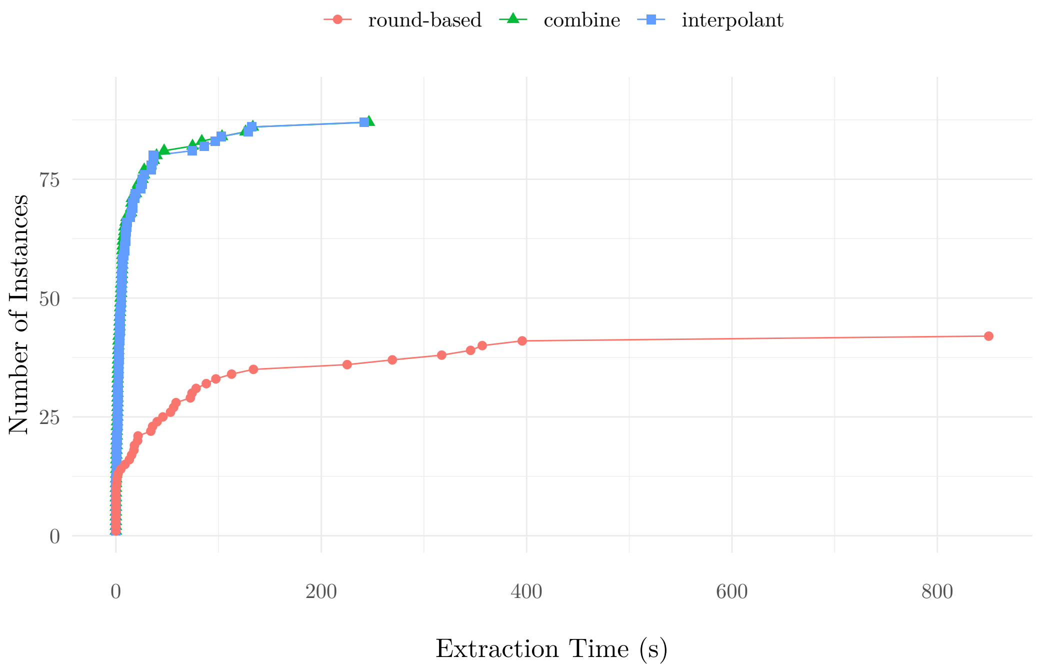


Figure 7 Number of strategies extracted within a given time, by algorithm.

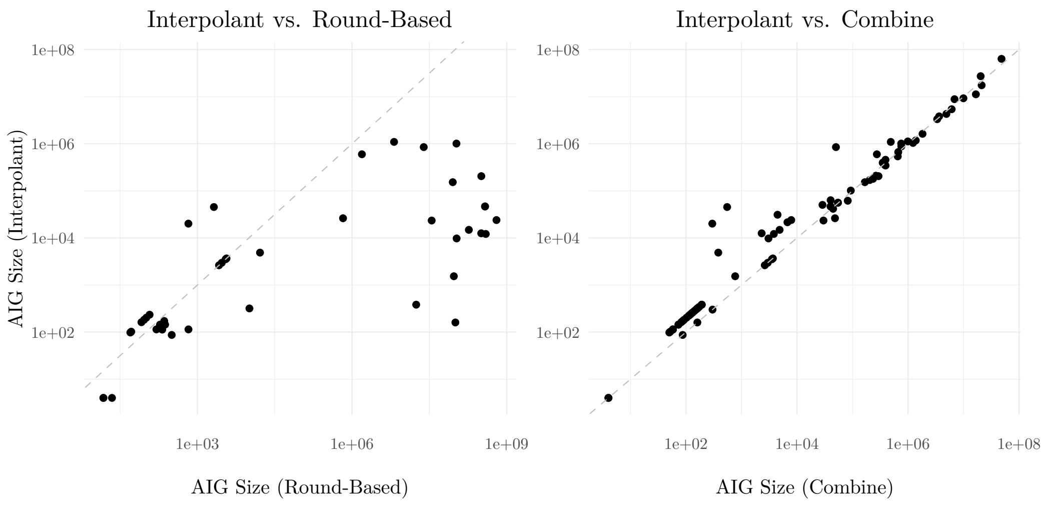


Figure 8 Comparison of AIG Sizes for extracted strategies (logarithmic scale).

5.3 Discussion

The gap between *round-based* and the other two strategy extraction algorithms seen in our results was expected. While the former performs multiple passes of the proof, one for each quantifier block, the other two only require a single pass. However, given that the three approaches are very different, it is surprising that *combine* and *interpolant* were consistently better: there was no instance where strategies could be extracted with *round-based*, but not with the other two algorithms, and the same is true for strategy verification. Even in terms of AIG *size*, there was no example where *round-based* resulted in smaller strategies than *combine* (there were a few instances where the AIGs were smaller with *round-based* compared to *interpolant*, however).

Another surprise was the performance of *combine* compared to *interpolant*. While both underlying algorithms have a running time of $O(mn)$ for a proof with m lines and n universal variables, a closer inspection shows that the hidden constant in this bound is about twice as large for *combine*. In spite of that, *combine* closely matched *interpolant*, frequently leading to smaller AIG sizes for strategies. One possible explanation for its good performance is that *combine* works with local strategies that are immediately substituted for universal variables, which in combination with hashing of AIG nodes may help compress strategies. By contrast, during the construction of circuit I_i for *interpolant*, universal variables u_j with $j < i$ are kept as inputs. Only at the very end, the interpolant I_i can be substituted for variable u_i.

6 Related Work

Goultiaeva et al. first observed that winning moves for the universal player in the QBF evaluation game can be efficiently extracted from Q-resolution refutations [18]. This result was generalised to long-distance Q-resolution by Egly et al. [17], and to IRM-calc by Beyersdorff et al. [5]. Efficient move extraction implies polynomial-time strategy extraction for proof systems that are closed under restriction. Peitl et al. gave an explicit construction for Q-resolution with a dependency scheme [32]. Balabanov and Jiang present a linear-time strategy extraction algorithm for Q-resolution [2] that was adapted to long-distance Q-resolution by Balabanov et al. [3]. Suda and Gleiss gave a local soundness argument for many resolution-based QBF proof systems, including Exp+Res [38]. They interpret clauses derived in these systems as abstractions of partial strategies, and show that resolution can be understood as an operation for combining partial strategies. Schlaipfer et al. used this interpretation of clauses as partial strategies, optimised for Exp+Res, to obtain an $O(mn)$ strategy extraction algorithm for a proof with m lines and n universal variables [34]. Chew and Slivovsky generalised this approach to prove simulations of many clausal QBF proof systems by extended QBF Frege [13].

Beyersdorff et al. lifted feasible interpolation as lower bound technique from propositional logic to QBF proof systems [6]. They also observed that interpolants and winning strategies coincide for the first universal variable in the quantifier prefix. Chew and Clymo extended this observation by proving that feasible interpolation of the underlying propositional proof system is necessary for polynomial-time strategy extraction in QBF expansion systems, and that interpolation is sufficient for polynomial-time strategy extraction whenever the propositional proof system is closed under restrictions [12].

Jiang et al. showed how to synthesise Boolean functions with a single output using interpolation [25]. Their approach can handle multiple outputs (i.e., multiple universal variables when applied to QBF strategy extraction) only by substituting functions and computing interpolants one at a time. Hofferek et al. extended their approach to multiple outputs and described an interpolation system that simultaneously extracts n interpolants from a single proof [20]. However, unlike the approach presented here, their interpolation system only works with ordered (so-called *local-first*) proofs, and transforming a proof into this shape may cause an exponential blowup.

7 Conclusion

This paper establishes a correspondence between strategy extraction, a key concept in QBF solving and proof complexity, and interpolation, a well studied technique in logic: every universal winning strategy of a QBF corresponds to a sequence of interpolants in its complete expansion, and vice versa. This observation inspired a new strategy extraction algorithm for QBF expansion proofs that performed well in our experiments. Correctness of this algorithm is proved here only for a specific (symmetric) interpolation system [33]. To assess the robustness of the correspondence between strategies and interpolants, it would be interesting to know whether the algorithm also works with other interpolation systems. Another followup question is whether it can be adapted to proof systems with partial annotations, such as IR-calc [5]. Finally, and perhaps most importantly, we conjecture that the main idea presented in this paper generalises beyond QBF to quantified SMT and instantiation-based first-order theorem proving, where it might find applications in complex synthesis tasks [20].

References

1 Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. An exponential separation between regular and general resolution. *Theory Comput.*, 3(1):81–102, 2007. doi:10.4086/TOC.2007.V003A005.

2 Valeriy Balabanov and Jie-Hong R. Jiang. Unified QBF certification and its applications. *Formal Methods Syst. Des.*, 41(1):45–65, 2012. doi:10.1007/S10703-012-0152-6.

3 Valeriy Balabanov, Jie-Hong Roland Jiang, Mikolas Janota, and Magdalena Widl. Efficient extraction of QBF (counter)models from long-distance resolution proofs. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 3694–3701. AAAI Press, 2015. doi:10.1609/AAAI.V29I1.9750.

4 Olaf Beyersdorff, Joshua Blinkhorn, Meena Mahajan, and Tomás Peitl. Hardness characterisations and size-width lower bounds for QBF resolution. *ACM Trans. Comput. Log.*, 24(2):10:1–10:30, 2023. doi:10.1145/3565286.

5 Olaf Beyersdorff, Leroy Chew, and Mikolás Janota. New resolution-based QBF calculi and their proof complexity. *ACM Trans. Comput. Theory*, 11(4):26:1–26:42, 2019. doi:10.1145/3352155.

6 Olaf Beyersdorff, Leroy Chew, Meena Mahajan, and Anil Shukla. Feasible interpolation for QBF resolution calculi. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:7)2017.

7 Armin Biere. Picosat essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008. doi:10.3233/SAT190039.

8 Armin Biere, Mathias Fleury, Nils Froleyks, and Marijn J. H. Heule. The SAT museum. In Matti Järvisalo and Daniel Le Berre, editors, *Proceedings of the 14th International Workshop on Pragmatics of SAT co-located with the 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Alghero, Italy, July 4, 2023*, volume 3545 of *CEUR Workshop Proceedings*, pages 72–87. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3545/paper6.pdf.

9 Roderick Bloem, Nicolas Braud-Santoni, Vedad Hadzic, Uwe Egly, Florian Lonsing, and Martina Seidl. Two SAT solvers for solving quantified boolean formulas with an arbitrary number of quantifier alternations. *Formal Methods Syst. Des.*, 57(2):157–177, 2021. doi:10.1007/S10703-021-00371-7.

10 Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, and Florian Lonsing. SAT-based methods for circuit synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 31–34. IEEE, 2014. doi:10.1109/FMCAD.2014.6987592.

11 Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation – 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2014. doi:10.1007/978-3-642-54013-4_1.

12 Leroy Chew and Judith Clymo. How QBF expansion makes strategy extraction hard. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning – 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 66–82. Springer, 2020. doi:10.1007/978-3-030-51074-9_5.

13 Leroy Chew and Friedrich Slivovsky. Towards uniform certification in QBF. In Petra Berenbrink and Benjamin Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 22:1–22:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.STACS.2022.22.

14 William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957. doi:10.2307/2963593.

15 Vijay Victor D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In Gilles Barthe and Manuel V. Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010. doi:10.1007/978-3-642-11319-2_{1}{2}.

16 Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 27(7):1165–1178, 2008. doi:10.1109/TCAD.2008.923410.

17 Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 291–308. Springer, 2013. doi:10.1007/978-3-642-45221-5_21.

18 Alexandra Goultiaeva, Allen Van Gelder, and Fahiem Bacchus. A uniform approach for generating proofs and strategies for both true and false QBF formulas. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 546–553. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-099.

19 Vedad Hadzic, Roderick Bloem, Ankit Shukla, and Martina Seidl. FERPModels: A certification framework for expansion-based QBF solving. In Bruno Buchberger, Mircea Marin, Viorel Negru, and Daniela Zaharie, editors, *24th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2022, Hagenberg / Linz, Austria, September 12-15, 2022*, pages 80–83. IEEE, 2022. doi:10.1109/SYNASC57785.2022.00022.

20 Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 77–84. IEEE, 2013. URL: https://ieeexplore.ieee.org/document/6679394/.

21 Guoxiang Huang. Constructing Craig interpolation formulas. In Ding-Zhu Du and Ming Li, editors, *Computing and Combinatorics, First Annual International Conference, COCOON '95, Xi'an, China, August 24-26, 1995, Proceedings*, volume 959 of *Lecture Notes in Computer Science*, pages 181–190. Springer, 1995. doi:10.1007/BFB0030832.

22 Mikolas Janota. On exponential lower bounds for partially ordered resolution. *J. Satisf. Boolean Model. Comput.*, 10(1):1–9, 2016. doi:10.3233/SAT190110.

23 Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016. doi:10.1016/J.ARTINT.2016.01.004.

24 Mikolás Janota and João Marques-Silva. Expansion-based QBF solving versus q-resolution. *Theor. Comput. Sci.*, 577:25–42, 2015. doi:10.1016/J.TCS.2015.01.048.

25 Jie-Hong Roland Jiang, Hsuan-Po Lin, and Wei-Lun Hung. Interpolating functions from large boolean relations. In Jaijeet S. Roychowdhury, editor, *2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009*, pages 779–784. ACM, 2009. doi:10.1145/1687399.1687544.

26 Laura Kovács and Andrei Voronkov. Interpolation and symbol elimination. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2009. doi:10.1007/978-3-642-02959-2_17.

27 Jan Krajícek. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. *J. Symb. Log.*, 62(2):457–486, 1997. doi:10.2307/2275541.

28 Paolo Mancosu. Introduction: Interpolations – Essays in honor of William Craig. *Synth.*, 164(3):313–319, 2008. doi:10.1007/S11229-008-9350-6.

29 Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003. doi:10.1007/978-3-540-45069-6_1.

30 Aina Niemetz, Mathias Preiner, Florian Lonsing, Martina Seidl, and Armin Biere. Resolution-based certificate extraction for QBF – (tool presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012 – 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 430–435. Springer, 2012. doi:10.1007/978-3-642-31612-8_33.

31 Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Polynomial-time validation of QCDCL certificates. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2018. doi:10.1007/978-3-319-94144-8_16.

32 Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Long-distance Q-resolution with dependency schemes. *J. Autom. Reason.*, 63(1):127–155, 2019. doi:10.1007/S10817-018-9467-3.

33 Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997. doi:10.2307/2275583.

34 Matthias Schlaipfer, Friedrich Slivovsky, Georg Weissenbacher, and Florian Zuleger. Multi-linear strategy extraction for QBF expansion proofs via local soundness. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020 – 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 429–446. Springer, 2020. doi:10.1007/978-3-030-51825-7_{3}{0}.

35 Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified boolean formulas. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*, pages 78–84. IEEE, 2019. doi:10.1109/ICTAI.2019.00020.

36 Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining BDDs. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science – Theory and Applications, First International Symposium on Computer Science in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006. doi:10.1007/11753728_60.

37 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006. doi:10.1145/1168857.1168907.

38 Martin Suda and Bernhard Gleiss. Local soundness for QBF calculi. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 217–234. Springer, 2018. doi:10.1007/978-3-319-94144-8_14.

39 Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, 103(11):2021–2035, 2015. doi:10.1109/JPROC.2015.2455034.

A Proof of Lemma 1

▶ **Lemma 1.** *Let $\Phi = Q_1 v_1 \dots Q_n v_n . \varphi$ be a QBF, and let $\alpha : \{v_1, \dots, v_i\} \to \{0,1\}$ be a partial assignment of its variables. Then $exp(\Phi[\alpha])$ and $\varphi^\sigma \wedge \tau^{[\sigma]}$ are equisatisfiable, where $\sigma = \alpha|_U$ and $\tau = \alpha|_E$.*

Proof. Let $E_i = \{v_1, \dots, v_i\} \cap E$ and $U_i = \{v_1, \dots, v_i\} \cap U$. By definition, we have

$$exp(\Phi[\alpha]) = \bigwedge_{\rho : U \setminus U_i \to \{0,1\}} \; \bigwedge_{C \in \varphi[\alpha]} C^{[\rho]}.$$

Since every remaining existential variable in a clause $C \in \varphi[\alpha]$ is to the right of every variable in the domain of σ, we can rename each literal $\ell^{[\rho]}$ in $exp(\Phi[\alpha])$ to $\ell^{[\rho \cup \sigma]}$ and obtain the equisatisfiable formula

$$\bigwedge_{\rho : U \setminus U_i \to \{0,1\}} \; \bigwedge_{C \in \varphi[\alpha]} C^{[\sigma \cup \rho]}.$$

After that, we can replace the conjunction over clauses $C \in \varphi[\alpha]$ with a conjunction over clauses $C \in \varphi[\tau]$, since $\alpha = \sigma \cup \tau$, and the effect of applying σ is taken care of by the instantiation. We thus get

$$\bigwedge_{\rho : U \setminus U_i \to \{0,1\}} \; \bigwedge_{C \in \varphi[\tau]} C^{[\sigma \cup \rho]} = \bigwedge_{\substack{\theta : U \to \{0,1\} \\ \sigma \subseteq \theta}} \; \bigwedge_{C \in \varphi[\tau]} C^{[\theta]} = \bigwedge_{\substack{\theta : U \to \{0,1\} \\ \sigma \subseteq \theta}} \; \bigwedge_{C \in \varphi} C[\tau]^{[\theta]}.$$

Because $var(\tau) \subseteq \{v_1, \dots, v_i\}$, $\tau^{[\theta]} = \tau^{[\sigma]}$ and

$$\bigwedge_{\substack{\theta : U \to \{0,1\} \\ \sigma \subseteq \theta}} \; \bigwedge_{C \in \varphi} C[\tau]^{[\theta]} = \bigwedge_{\substack{\theta : U \to \{0,1\} \\ \sigma \subseteq \theta}} \; \bigwedge_{C \in \varphi} C^{[\theta]} \wedge \tau^{[\sigma]} = \varphi^\sigma \wedge \tau^{[\sigma]}.$$ ◀

Quantum Circuit Mapping Based on Incremental and Parallel SAT Solving

Jiong Yang
National University of Singapore, Singapore

Yaroslav A. Kharkov
AWS Quantum Technologies, New York, NY, USA

Yunong Shi
AWS Quantum Technologies, New York, NY, USA

Marijn J. H. Heule
Carnegie Mellon University, Pittsburgh, PA, USA
Amazon Web Services, Seattle, WA, USA

Bruno Dutertre
Amazon Web Services, Santa Clara, CA, USA

Abstract

Quantum Computing (QC) is a new computational paradigm that promises significant speedup over classical computing in various domains. However, near-term QC faces numerous challenges, including limited qubit connectivity and noisy quantum operations. To address the qubit connectivity constraint, circuit mapping is required for executing quantum circuits on quantum computers. This process involves performing initial qubit placement and using the quantum SWAP operations to relocate non-adjacent qubits for nearest-neighbor interaction. Reducing the SWAP count in circuit mapping is essential for improving the success rate of quantum circuit execution as SWAPs are costly and error-prone. In this work, we introduce a novel circuit mapping method by combining incremental and parallel solving for Boolean Satisfiability (SAT). We present an innovative SAT encoding for circuit mapping problems, which significantly improves solver-based mapping methods and provides a smooth trade-off between compilation quality and compilation time. Through comprehensive benchmarking of 78 instances covering 3 quantum algorithms on 2 distinct quantum computer topologies, we demonstrate that our method is 26× faster than state-of-the-art solver-based methods, reducing the compilation time from hours to minutes for important quantum applications. Our method also surpasses the existing heuristics algorithm by 26% in SWAP count.

2012 ACM Subject Classification Software and its engineering → Compilers; Computer systems organization → Quantum computing; Computing methodologies → Artificial intelligence

Keywords and phrases Quantum computing, Quantum compilation, SAT solving, Incremental solving, Parallel solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.29

Funding *Marijn J. H. Heule*: Supported by the NSF under grant CCF-2229099.

Acknowledgements Part of the research was conducted during Jiong Yang's internship at Amazon Web Services. We thank Eric Kessler and Soonho Kong for the insightful discussions on this work. We are grateful to the anonymous reviewers for their constructive comments to improve this paper.

1 Introduction

There is compelling evidence that Quantum Computing (QC) can solve certain computational problems exponentially more efficiently than classical computers [33, 39]. As a result, transformative applications are expected to emerge in fields such as optimization [1, 11, 32], machine learning [8, 38], finance [10, 18, 21, 34], pharmaceuticals [13, 16, 24], and cryptography [17, 39, 40].

However, the practical realization of QC still faces numerous challenges, including limited qubit connectivity on quantum computers and noisy quantum operations. Limited qubit connectivity decreases the ability of quantum devices to execute arbitrary quantum circuits, and noisy operations restrict the sizes of executable quantum circuits. Fortunately, these hardware challenges can be mitigated at the software level by compiler optimizations.

Quantum compilers perform numerous transformations and optimizations to produce compact and optimized circuit executables. The specific transformation we are concerned with in this paper is *circuit mapping*. Circuit mapping involves the insertion of a special quantum operation called the SWAP gate to map arbitrary quantum circuits to devices. Since SWAP gates are costly and error-prone, the compiler must minimize the SWAP count. Currently, there are two primary approaches to the circuit mapping problem: solver-based algorithms [28, 44] and heuristics-based algorithms [26, 41]. Both approaches have their drawbacks: solver-based algorithms achieve optimal SWAP count but suffer from long compilation time; heuristic algorithms are fast, but the SWAP counts are usually suboptimal.

We propose a novel circuit mapping method based on incremental and parallel solving for Boolean Satisfiability (SAT). Our approach aims to find a minimum number of SWAP gates that accommodate the circuit mapping requirement by iteratively decreasing the SWAP-gate count and checking feasibility with SAT solving. We use a dedicated SAT encoding that enables incremental solving and we combine incremental and parallel solving techniques. Compared to current solver-based algorithms, our method is 26x faster on average. Compared to current heuristic algorithms, our method reduces the SWAP count by 26% on average.

In summary, our contributions are:

1. We design a novel SAT encoding for determining the satisfiability of mapping a circuit with a given SWAP count.
2. By combining the novel SAT encoding and parameter search developed, we develop a new circuit mapping method that achieves a smooth trade-off between compilation quality (in terms of SWAP count) and compilation time.
3. By exploiting the problem structure, we develop an efficient implementation of the proposed mapping method that combines incremental and parallel techniques.
4. We perform an extensive evaluation to show that the resulting approach is 26× faster than the state-of-the-art solver-based method and outperforms the heuristic approaches in 76% of the instances.

In the rest of this paper, Section 2 introduces the background on quantum circuit mapping and preliminaries on SAT solving. We then present our SAT-based quantum circuit mapper in Section 3 and give details on the full encoding in Section 4. Finally, we present our experimental evaluation in Section 5 and conclude in Section 6.

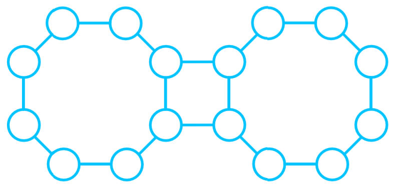
2 Background

2.1 Quantum Circuit Mapping

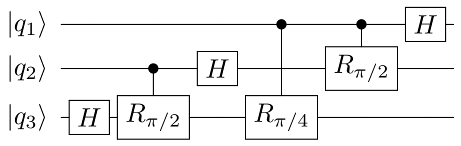
We first illustrate the problem of quantum circuit mapping with an example. Subsequently, we delve into the existing approaches addressing this problem.

2.1.1 An Illustrative Example

Figure 1a shows the qubit connectivity of (part of) the Aspen M-3 quantum computer, manufactured by Rigetti Computing [36]. Figure 1b shows the circuit diagram of a 3-qubit quantum circuit for the famous Quantum Fourier Transform (QFT) algorithm [14, 33]. The QFT circuit has a 2-qubit gate between each pair of the three qubits.



(a) Qubit connectivity graph of (part of) the Rigetti Aspen M-3 quantum computer. Circles are physical qubits and lines are physical links that allow 2-qubit gates to be performed on.

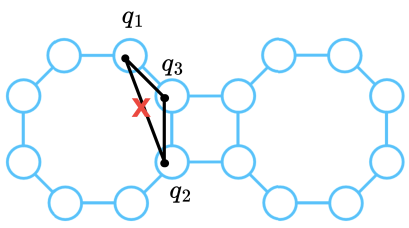


(b) An example 3-qubit QFT circuit. The 3 horizontal lines represent the time schedules of the quantum gates on the 3 algorithmic qubits q_1, q_2, and q_3. This circuit consists of 1-qubit Hadamard gates (boxes labeled H) and 2-qubit controlled rotation gates (solid dot and box labeled R that are connected by a vertical line). In our problem setting, the two qubits in controlled rotation gates can be treated equally.

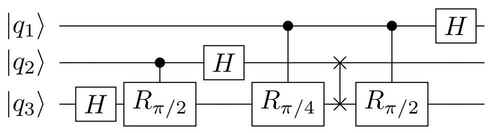
Figure 1 The mismatch between qubit connectivity of the Rigetti Aspen M-3 hardware and that of the QFT circuit. On the Aspen M-3 device in Figure 1a, no three qubits are connected to each other. In the QFT circuit in Figure 1b, each qubit is connected to the other two by a 2-qubit gate.

To execute the QFT circuit on the Aspen M-3 device, a *circuit mapping* procedure must be performed by a quantum compiler. Circuit mapping involves two steps: initial qubit placement and qubit routing. During initial qubit placement, the quantum compiler maps each algorithmic qubit in the circuit to a physical qubit on the device, as shown in Figure 2a.

The QFT circuit requires each algorithmic qubit to interact with the other two. However, on the qubit connectivity graph of Aspen M-3, no subgraph forms a three-qubit ring that would allow pairwise qubit interaction. As a result, after initial qubit placement, the QFT circuit still cannot be directly executed since it requires a non-local 2-qubit interaction that is not supported by the device (see Figure 2a for a specific initial assignment). This gap necessitates the second step in circuit mapping – *qubit routing*. When two (algorithmic) qubit operands in a 2-qubit gate are mapped to non-adjacent physical qubits, the compiler performs qubit routing to remap the two qubit operands to adjacent physical qubits before scheduling the 2-qubit gate.



(a) The initial qubit placement step associates each algorithmic qubit in the QFT circuit with a physical qubit on the Aspen M-3 device. Black lines are connections required by the QFT circuit in Figure 1b. The line with red cross is not available on the device.



(b) The qubit routing step inserts a SWAP gate (denoted as two crosses connected by a vertical line) in the QFT circuit. After the SWAP insertion, the last 2-qubit gate is now between q_1 and q_3 (it was between q_1 and q_2, as shown in Figure 1b).

Figure 2 Mapping the 3-qubit QFT circuit in Figure 1b onto the Aspen M-3 device in Figure 1a. First, algorithmic qubits are placed onto the device (Figure 2a). The last 2-qubit gate between q_1 and q_2 cannot be scheduled as there is no link between them on the device.. After the SWAP insertion (Figure 2b), the last 2-qubit gate is re-targeted to q_1 and q_3, which then can be scheduled.

Qubit routing is performed by inserting quantum SWAP gates [33]. A SWAP gate is a special 2-qubit quantum gate that is not responsible for entangling qubit states for computation, but for exchanging the qubit states for routing. Since a SWAP gate exchanges qubit states, it affects the quantum gates scheduled behind it. After a SWAP insertion, the gates scheduled after the SWAP gate that use one of the swapped qubits must be re-targeted to the other qubit that it swaps to. Figure 2b gives an example of SWAP insertion for the 3-qubit QFT circuit from Figure 1b. From the example, we can see that SWAP gates can alter the connectivity requirements of the quantum circuits to match them with the qubit connectivity of the underlying quantum hardware.

2.1.2 Current Circuit Mapping Approaches

Currently, there are two main-stream approaches to the circuit mapping problem:

Heuristic Algorithms. Heuristic mapping algorithms usually optimize metrics designed by humans and calculable within a bounded search depth. Examples of these metrics include the total 2-qubit gate distance of the 2-qubit gates that remain t to be scheduled (the gate distance of a 2-qubit gate measures how far away the two qubit operands are on the device given the current mapping). In industrial quantum compilers, heuristic algorithms are among the most popular choices because they provide fast compilation time. Notable examples include the SABRE (SWAP-based BidiREctional heuristic search) algorithm [26] used in the Qiskit compiler [2] and the architecture-aware mapping algorithm in the TKET compiler [41]. Optimality studies have shown that heuristic algorithms are far from the theoretical optimal in terms of the output SWAP count [43].

Solver-based Algorithms. Alternatives to the heuristic methods rely on complete algorithms to search for the minimal SWAP count. Existing work employs different types of solvers for this purpose. The state-of-the-art solver-based algorithm is the TB-OLSQ2 mapper [28], which translates the mapping problem to a satisfiability problem that can be solved by the Z3 [15] SMT solver. SATMap [31] is another solver-based algorithm that is based on MaxSAT solving but SATMap is not as efficient as TB-OLSQ2 due to a different choice of encoding.

2.2 Modern SAT Solving

Let $\{x_1, x_2, ..., x_n\}$ be a set of Boolean variables. A literal l is a variable x or its negation $\neg x$. A clause C of size k is a disjunction of k literals, i.e., $C = (l_1 \vee l_2 \vee ... \vee l_k)$. A formula φ in Conjunctive Normal Formal (CNF) is a conjunction of clauses, i.e., $\varphi = (C_1 \wedge C_2 \wedge ... \wedge C_m)$ for some $m > 0$. An assignment of truth values to the variables in φ is called a *solution* if it makes φ evaluate to true. We call φ *satisfiable* if there exists a solution and *unsatisfiable* otherwise. Given a formula φ, modern SAT solvers can effectively find a solution or prove that it is unsatisfiable by using the Conflict Driven Clause Learning (CDCL) scheme [29]. Many efficient solver implementations have emerged in recent years, including but not limited to Kissat, CaDiCaL [9], CryptoMiniSat [42], MapleSAT [27], and Glucose [5].

Incremental Solving. Incremental solving is an effective technique supported by most modern SAT solvers for solving a series of similar formulas. Incremental solving supports the addition of new clauses and assumptions between SAT calls. A key benefit is that the solver can keep and reuse internal states and clauses learned in previous SAT calls to speed up solving for the new formula.

Parallel Solving. Parallel solving seeks to distribute the computational workload of solving a formula across multiple processors. There are primarily two categories of work. The *partition* approaches try to split the formula equally into many sub-problems using heuristics and solve them in parallel [4, 20, 23, 45]. Recent work in this line includes cube-and-conquer [23], AmPharoS [4], and Paracooba [20]. Another line of work, called the*portfolio* approach, runs multiple SAT solvers with different configurations in parallel to solve the original formula and share information, such as, learned clauses between solvers. These approaches mainly leverage the solver diversity to improve the overall solving speed. Portfolio approaches are implemented by competition-winning solvers such as ManySAT [19], Mallob [37], and ParKissat-RS [7].

2.3 Helper Functions

In this section, we introduce some of the helper functions that we use in our approach.

2.3.1 Linear Encoding of At-Most-One Constraints

Several constraints in our SAT formulation are *At-Most-One* (AMO) constraints. We leverage a recursive scheme (also used in other works [22]), as a general helper function for encoding AMO constraints in our SAT formulation. Each step of the recursion introduces a new Boolean variable y. To encode the AMO constraint for a general set of Boolean variables $B = \{b_1, ..., b_i, ..., b_n\}$, we have,

$$\mathrm{AMO}(b_1, \dots, b_n) = \begin{cases} \bigwedge_{1 \le i < j \le n} (\neg b_i \vee \neg b_j) & n \le 4 \\ \mathrm{AMO}(b_1, b_2, b_3, y) \wedge \mathrm{AMO}(\neg y, b_4, \dots, b_n) & \text{otherwise} \end{cases}$$

For a constraint with n variables, this encoding introduces $\frac{n-3}{2}$ auxiliary variables and $3(n-2)$ clauses compared to $\frac{n^2-n}{2}$ clauses using a naive pairwise encoding.

2.3.2 Totalizer Encoding

Our approach utilizes a cardinality constraint with an iteratively decreasing cardinality bound. We hence describe the Totalizer encoding [6, 30] for cardinality constraints with varying bounds. Below we present an example to encode the AtMostK constraint $(l_1+l_2+l_3+l_4+l_5 \le k)$. Figure 3 illustrates the encoding structure where every node is represented by a node name and a list of variables. The leaf node indicates an input variable, e.g., node C with a variable l_1. The root node includes the indicator variables for the sum of input variables. For example, assigning variable o_2 in node O indicates that at least two variables of $l_1, ..., l_5$ are true, while a false assignment to o_2 indicates that at most one input variable is assigned to true. Every internal node represents an intermediate sum over its two children. For example, node A represents the sum of variables from nodes C and D. The variables a_1 and a_2 indicate whether the sum is at least one and two, respectively.

Following the structure, we can derive the encoding below. For any non-leaf node $P : p_1, ..., p_{n_p}$ with two children denoted by $Q : q_1, ..., q_{n_q}$ and $R : r_1, ..., r_{n_r}$, we require at least $\alpha + \beta$ variables of P to be true when node Q implies α many variables assigned to true and node R indicates β many variables to be true, i.e.,

$$\begin{aligned} &(q_\alpha \wedge r_\beta) \rightarrow p_\gamma \\ \iff &\neg q_\alpha \vee \neg r_\beta \vee p_\gamma \quad \text{for } \alpha + \beta = \gamma,\ 0 \le \alpha \le n_q,\ 0 \le \beta \le n_r,\ 0 \le \gamma \le n_p \end{aligned}$$

$$\begin{tikzcd}
 & & & & O : o_1, o_2, o_3, o_4, o_5 \arrow[dlll] \arrow[drr] & & & & \\
 & A : a_1, a_2 \arrow[dl] \arrow[dr] & & & & & B : b_1, b_2, b_3 \arrow[dl] \arrow[dr] & & \\
C : l_1 & & D : l_2 & & & E : l_3 & & F : f_1, f_2 \arrow[dl] \arrow[dr] & \\
 & & & & & & G : l_4 & & H : l_5
\end{tikzcd}$$

Figure 3 Totalizer encoding for $l_1 + l_2 + l_3 + l_4 + l_5 \leq k$.

where $q_0 = r_0 = p_0 =$ true. For the cardinality bound, we simply add a unit clause

$$\neg o_{k+1}$$

for AtMostK constraint or add o_k for AtLeastK constraint. In our application, we are interested in AtMostK constraints with iteratively decreasing bounds. Since $\neg o_{k_1}$ implies $\neg o_{k_2}$ for $k_1 < k_2$, we can add the unit clause for a smaller bound without deleting the previous one, which is called *incremental strengthening* in the literature [3].

For a cardinality constraint with n variables and bound k, the Totalizer encoding requires $\mathcal{O}(n \log n)$ auxiliary variables and $\mathcal{O}(nk)$ clauses after simplification [12, 25].

3 SAT-based Circuit Mapping

Our novel quantum circuit mapper is called SATmapper. It is depicted in Figure 4. SATmapper seeks to find a minimum number of SWAP gates that accommodate the quantum circuit mapping requirement by iteratively decreasing the SWAP-gate count (S) and checking feasibility for S using a modern SAT solver. Section 3.1 provides an overview of the SATmapper framework. We then discuss the encoding and decoding processes in Section 3.2. We discuss the solving techniques used by SATmapper in Section 3.3.

3.1 Framework

Figure 4 presents the framework of SATmapper. For a given number of SWAP count S, SATmapper reduces the mapping problem to a SAT encoding and utilizes a SAT solver to compute the feasibility of using no mote than S SWAP gates. The workflow runs from left to right. Given three inputs: a quantum circuit, a quantum device (QPU), and an initial SWAP count (S), SATmapper encodes the quantum circuit mapping problem into a SAT formula in conjunctive normal form (CNF). A SAT solver takes the CNF as input and checks its satisfiability. A satisfiable (SAT) result indicates that there is a valid mapping that uses no more than S SWAP gates. In this case, we reduce the SWAP count S and continue the loop to search for a mapping with fewer SWAP gates. We exit the loop when the solver returns UNSAT, which indicates that we cannot decrease the SWAP count. Finally, we decode a mapped circuit from the best result we've obtained so far.

Pseudo-code for this procedure is given in Algorithm 1 . Given a quantum circuit and a device. we initialize S with the value produced by the state-of-the-art heuristic approach, TKET. If TKET generates a mapping without any SWAP gates, we terminate and return the mapping at Line 3. In the common case, TKET provides a mapping with $S(> 0)$ SWAP gates. Then, we start our SAT-based optimization from this S at Lines 4-11.

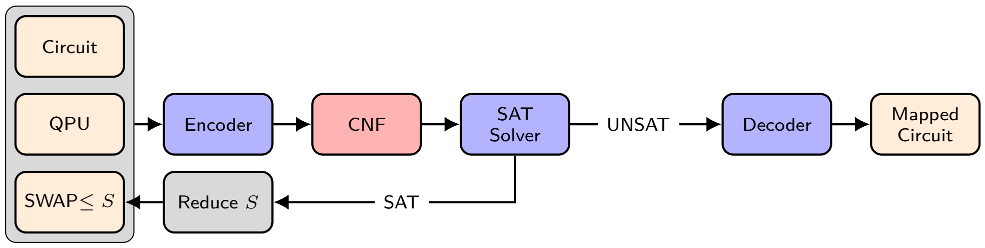


Figure 4 SATmapper framework.

Algorithm 1 SATmapper(circuit, qpu).

```
1:  $S \leftarrow$ TKET(circuit, qpu);
2:  if $S = 0$ then
3:      return mapped circuit from TKET;                     ▷ It's already optimal.
4:  repeat
5:      $S \leftarrow S - 1$;
6:      $\varphi \leftarrow$ Encode(circuit, qpu, $S$);
7:      sat $\leftarrow$ Solve($\varphi$);
8:      if sat is True then
9:          solution $\leftarrow$ GetSolution($\varphi$);             ▷ Store the best result so far.
10:         $S \leftarrow$ CountSWAP(solution);       ▷ Calculate the SWAP count in the solution.
11: until (sat is False) or ($S = 0$) or Timeout;
12: if solution is NULL then
13:     return mapped circuit from TKET;            ▷ No better result from SATmapper.
14: else
15:     mappedCircuit $\leftarrow$ Decode(solution, circuit, qpu);
16:     return mappedCircuit;              ▷ Retrieve the best result within the time limit.
```

In every iteration, we reduce the best-known S by one to explore the possibility of producing a lower SWAP count. We encode the circuit mapping with an upper bound of S SWAPs into a formula φ at Line 6 and invoke a SAT solver to solve φ at Line 7. If φ is satisfiable, the circuit can be mapped with no more than S SWAP gates and we store the solution at Line 9. The solution contains the best mapping up to this point and can later be decoded into a mapped circuit. We calculate the actual SWAP count implied by the solution as this SWAP count can be smaller than the current upper bound S. Therefore, we update the best-known SWAP count S with the actual value implied by the solution at Line 10. We repeat the loop until the solver can't find a better mapping, or S is zero, or we exceed the time limit. If we can't find a single solution in the loop, we return the initial mapping from TKET at Line 13. Otherwise, a solution was stored and we decode at Line 15 a mapped circuit from this solution, which represents the mapping with the smallest SWAP count obtained within the time limit.

3.2 Encoding Sketch

This section presents the key encoding idea for mapping a quantum circuit to a target device using a given number of SWAP gates. We introduce the core variables that encode the basic information for mapping circuits in Section 3.2.1. Next, Section 3.2.2 presents the encoding of SWAP constraints which has a crucial impact on our system design in the following sections.

Other variables and constraints capture the mapping interaction between the circuit and SWAP gates, and we defer the details of full encoding to Section 4. Finally, Section 3.2.3 illustrates how to decode the mapped circuit from a solution to our encoding.

3.2.1 Transition Step

We consider the circuit mapping in different temporal steps. Instead of the actual time step in the circuit that increase gate by gate, we define a *transition* step that increases only after SWAP gate insertion. The layout of a quantum device changes when the transition step increases. Each layout update event is encoded as a new time step. Therefore, we start with an initial layout at step 0 and move to step 1 after inserting SWAP gates to update the layout. Given a qubit connectivity graph of a quantum device, we use a Boolean variable c_k^t for every positive step t and edge k to indicate whether the edge k is selected to perform a SWAP at step t or not. c_k^t is assigned to true if the edge is selected to perform a SWAP and is assigned to false otherwise.

For every gate g and step t, we use a Boolean variable o_g^t to indicate whether the gate g has been scheduled by step t (including t) or not. If o_g^t is assigned to true, gate g is scheduled at step i with $0 \leq i \leq t$ and is assigned to false otherwise. For example, $o_g^1 =$ True indicates that the gate g has been scheduled by step 1, that is, scheduled at either step 0 or step 1. Otherwise, o_g^1 is false and the gate g has to be scheduled at step $t > 1$.

3.2.2 SWAP Constraints

The constraint on SWAP count is crucial to our design of SATmapper as it is the only varying constraint across iterations. Suppose we consider T transition steps and K edges of the connectivity graph. During the transition, multiple SWAPs can be scheduled simultaneously if there are no conflicts among them. For every step $t > 0$, adjacent edges can't be selected for SWAP at the same time because they share a common vertex, For any two edges k and k' sharing a common vertex, we assert then

$$\neg c_k^t \vee \neg c_{k'}^t \text{ for } t \in [1, T]$$

The total number of SWAPs over T steps is at most S, i.e.,

$$\sum_{t=1}^{T} \sum_{k=1}^{K} c_k^t \leq S \tag{1}$$

where c_k^t indicates whether the edge k is selected to perform SWAP at step t. The cardinality constraint (Equation 1) can be encoded into clauses using the Totalizer encoding detailed in Section 2.3.2.

3.2.3 Mapped Circuit Decoding

Algorithm 2 describes how to decode a mapped circuit from a solution to our encoding. We first decode an initial qubit placement from the solution, that is, the mapping from each algorithmic qubit to a physical qubit at step 0. Then, we retrieve the number of transition steps from the heuristic approach, TKET. Starting from Line 3, we decode the mapped circuit step by step. We use a list l initialized at Line 4 to store the gates to be scheduled in sequence. Except for the initial step ($t = 0$), we retrieve all SWAP gates at step t and schedule them at Line 8 to update the layout. Since there is no conflict between the SWAP

Algorithm 2 Decode(solution, circuit, qpu).

```
1:  mappedCircuit ← GetInitPlacement(solution);
2:  $T$ ← TKETstep(circuit, qpu);
3:  for $t \leftarrow 0$ to $T$ do
4:      $l \leftarrow$ an empty list;
5:      if $t > 0$ then
6:          for $c_k^t$ from solution do
7:              if $c_k^t$ is True then
8:                  mappedCircuit adds a SWAP gate on edge $k$;
9:      for $o_g^t$ from solution do
10:         if $o_g^t$ is True then
11:             for $g' \leftarrow$ the last to the first of $l$ do
12:                 if $g$ depends on $g'$ in circuit then
13:                     $l$ inserts $g$ after $g'$;
14:                     break
15:             if $g \notin l$ then
16:                 $l$ inserts $g$ at the beginning;
17:     for $g \in l$ do
18:         mappedCircuit adds $g$;
19: return mappedCircuit;
```

gates at the same step, they can be scheduled in any order. Following that, we retrieve other gates scheduled at step t one by one starting from Line 9. For a gate g, we traverse the list l in a *reverse* order (Line 11) to insert g after the last gate g' that g depends on at Line 13, which ensures g is scheduled after all dependent gates. If g has no dependent gates in l, we insert g at the beginning of l at Line 16. After that, l contains a sequence of gates where $\forall g_i, g_j \in l$ with $i < j$, g_i doesn't depend on g_j and can be safely scheduled before g_j. Finally, we schedule every gate in l in sequence at Line 18 and return the mapped circuit at Line 19.

3.3 Solving Techniques

Any modern SAT solver can serve to solve the encoded CNF in Figure 4, such as Kissat, CaDiCaL [9], and the like. To embrace the recent advance in parallel (cloud) SAT solving, we can use a competition-winning parallel SAT solver instead like Mallob [37] or ParKissat-RS [7]. Furthermore, Section 3.3.1 presents the application of incremental solving techniques to SATmapper. Finally, we introduce the combination of incremental and parallel solving in Section 3.3.2 to benefit SATmapper from the best of both worlds.

3.3.1 Incremental Solving

Incremental solving is beneficial in solving a series of similar input formulas. For SATmapper, only the AtMostK cardinality constraint on the SWAP count (Equation 1) varies over iterations. According to Section 2.3.2, the Totalizer encoding of the AtMostK constraint allows us to use a list of variables o_i to indicate different upper bounds for the same encoding. In every iteration of SATmapper, we can run an incremental SAT solver, for example, CaDiCaL, with a different assumption. For example, we can impose Equation 1 in our encoding by adding the assumption of $\neg o_{S+1}$ to the SAT solver, where $\neg o_{S+1}$ indicates that there won't be $S + 1$ input variables to be true, that is, at most S SWAP gates are allowed in the mapping. When SATmapper moves to a lower SWAP count S' with $S' < S$, we add another assumption $\neg o_{S'+1}$ to indicate the new bound of no more than S' SWAPs.

As a consequence, we can avoid encoding and reading the whole CNF at every iteration on line 6 and 7 of Algorithm 1. Instead, the solver uses a new assumption literal and continues to solve the formula. This incremental solving allows the solver to start from the previous internal state and reuse learned clauses from previous iterations, which effectively reduces the overall runtime across iterations.

3.3.2 Incremental Parallel Solving

Given the application of incremental or parallel solving to SATmapper individually, one would wonder whether SATmapper can benefit from the best of both worlds. This section gives an affirmative answer to this question by presenting an incremental and parallel SAT solver, IncParKissat.

We have extended the state-of-the-art parallel SAT solver ParKissat-RS to support incremental solving. Figure 5 depicts the framework of IncParKissat. IncParKissat takes an original CNF as input and initially forks multiple Kissat with different configurations to run in parallel. Each Kissat keeps an individual copy of the input CNF and maintains its clause database throughout running. During the solving process, different solvers communicate with each other by sending and receiving important learned clauses to facilitate the global solving. When a solver finds a solution, a global termination signal is sent to the other solvers, and all solver threads join the main thread. Alternatively, if a solver proves that the input CNF is UNSAT, it terminates and joins the main thread. In either case, we output the result into the main thread and continue the incremental solving by forking multiple solver threads again. Every solver adds additional clauses φ to their database and continues to solve the input formula from the previous internal state.

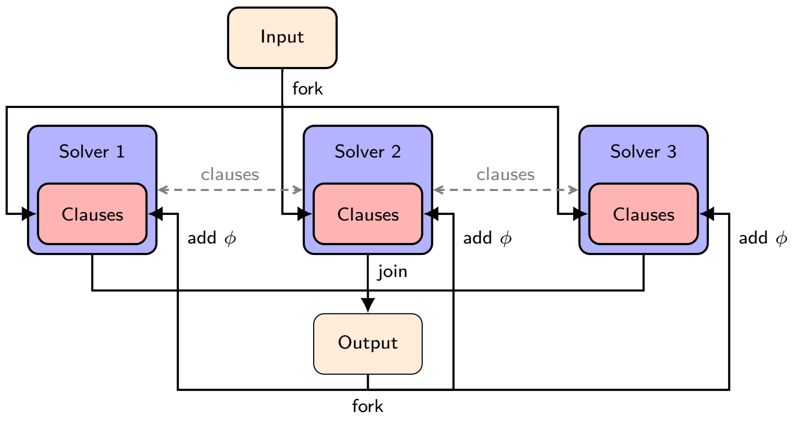


Figure 5 Framework of incremental and parallel SAT solver, IncParKissat.

The underlying solver Kissat used by IncParKissat does not currently support solving with assumptions, but our application, assumptions are not required. Our implementation allows IncParKissat to add extra clauses across incremental iterations. Because SATmapper only requires a cardinality constraint with a decreasing upper bound across iterations, the ability to add a unit clause across iterations is sufficient. This is because $\neg o_{k_1}$ implies $\neg o_{k_2}$ for $k_1 < k_2$ in the Totalizer encoding, so we can add the unit clause for a smaller bound without deleting the previous one, as described in Section 2.3.2. IncParKissat could be extended to support solving with assumptions by replacing the underlying SAT solver Kissat with CaDiCaL.

4 Full SAT Encoding

We present the full SAT encoding for the quantum circuit mapping problem in this section. Our encoding uses a *transition*-based step instead of the actual time steps in the circuit. After SWAP insertion, the layout of the quantum device is updated and we call the layout update a *transition*. Each layout transition event will be encoded as a new time step. The decision of whether a gate has been scheduled by a layout transition or not is encoded as a Boolean variable.

As follows, Section 4.1 introduces the formal definition for input and Section 4.2 discusses the pre-processing steps. We define the encoding variables in Section 4.3 and constraints in Section 4.4. We analyze the asymptotic encoding size in Section 4.5.

4.1 Problem Input

There are three inputs to the SAT formulation: (1) The quantum circuit to be mapped, which is represented as an ordered list of n 2-qubit[1] quantum gates $G = \{g_1, ..., g_i, ..., g_n\}$ and V algorithmic qubits; (2) The qubit connectivity graph of the quantum device, including P physical qubits and K connecting edges e_{ij} for $i, j \in [1, P]$ and $i \neq j$; (3) the desired physical SWAP gate number S.

4.2 Pre-processing

We perform two simple initialization steps as pre-processing of the SAT encoding:

- **Initialization of the input qubit connectivity graph.** We initialize an ordered list $E = \{d_1, ..., d_k, ..., d_K\}$ of edges, where each d_k uniquely corresponds to an edge $e_{i_k j_k}$ in the input qubit connectivity graph, and K is the total number of edges in the connectivity graph. The order of edges e_{ij} in E can be chosen arbitrarily. For each d_k, we denote the larger qubit index it connects as $d_k.op_{\max}$ and the smaller qubit index as $d_k.op_{\min}$.
- **Initialization of gate dependency list.** For g_i in G, we denote the larger gate operand as $g_i.op_{\max}$ and the smaller operand as $g_i.op_{\min}$ and initialize them in an array. Both $g_i.op_{\max}$ and $g_i.op_{\min}$ are fixed for $i \in [1, n]$. Further, we initialize the gate dependency list $l_g = \{(g_1^1, g_2^1), ..., (g_1^i, g_2^i), ...\}$, each pair of gates denotes that g_2^i is dependent on g_1^i and thus g_2^i cannot be scheduled before g_1^i. l_g can be generated by enumerating the gates on each qubit.

4.3 Encoding Variables

- Schedule $o_{g_i}^t$: if gate g_i has been scheduled in step t, $t \in [0, T]$, then $o_{g_i}^t$ = True, otherwise, $o_{g_i}^t$ = False. 2-qubit gates can only be scheduled after their algorithmic qubit operands are mapped to connected physical qubits on the connectivity graph. We set the number of transition steps T to be the same as the output of the heuristic approach, TKET.
- Layout π_{ij}^t: After step t (and before step $t+1$ if $t < T$), if qubit i is mapped to qubit j, then π_{ij}^t = True, else π_{ij}^t = False. Here $i \in [1, V], j \in [1, P], t \in [0, T]$.
- SWAP operand selection c_k^t: if edge $k \in [1, K]$ is selected for performing a SWAP at step $t \in [1, T]$, then c_k^t = True, otherwise, c_k^t = False. There are no SWAPs in the initial step, when $t = 0$.

[1] We don't consider 1-qubit gates because they can always be scheduled.

4.4 Constraints

- **Gate schedule initialization.** For step $t \in [0, T)$ and gate $g_i \in G$, it's impossible that g_i is scheduled after step t, but not scheduled after step $t+1$, i.e.,

$$\begin{aligned} &\neg(o^t_{g_i} \wedge \neg o^{t+1}_{g_i}) \\ \Longleftrightarrow &\neg o^t_{g_i} \vee o^{t+1}_{g_i} \text{ for } i \in [1, n],\ t \in [0, T) \end{aligned}$$

  by de Morgan's law. Also, all gates should be scheduled after step T, i.e.,

$$o^T_{g_i} = \text{True for } i \in [1, n]$$

- **Gate dependency.** For $(g_i, g_j) \in l_g$, g_j cannot be scheduled before g_i:

$$\neg(\neg o^t_{g_i} \wedge o^t_{g_j}) \text{ for } (g_i, g_j) \in l_g, \text{ and } t \in [0, T)$$

  similarly, by de Morgan's law, it can be re-written as

$$o^t_{g_i} \vee \neg o^t_{g_j} \text{ for } (g_i, g_j) \in l_g, \text{ and } t \in [0, T)$$

- **Gate schedule continuation.** For all $i \in [1, n]$ and $t \in [1, T]$, if gate g_i is not scheduled after step $t-1$ and its qubit operands are not mapped to connected physical qubits after step t, then g_i will not be scheduled after step t.

$$\left(\neg o^{t-1}_{g_i} \wedge \bigwedge_{(j,l) \in E} \neg \left((\pi^t_{g_i.op_{\max}j} \wedge \pi^t_{g_i.op_{\min}l}) \vee (\pi^t_{g_i.op_{\max}l} \wedge \pi^t_{g_i.op_{\min}j}) \right) \right) \rightarrow \neg o^t_{g_i}$$

  for $i \in [1, n]$, and $t \in [1, T]$, which can be re-written as

$$o^{t-1}_{g_i} \vee \neg o^t_{g_i} \vee \bigvee_{(j,l) \in E} (\pi^t_{g_i.op_{\max}j} \wedge \pi^t_{g_i.op_{\min}l}) \vee (\pi^t_{g_i.op_{\max}l} \wedge \pi^t_{g_i.op_{\min}j})$$

  where the big disjunction can be resolved by introducing auxiliary variables. For $t = 0$,

$$\neg o^0_{g_i} \vee \bigvee_{(j,l) \in E} (\pi^0_{g_i.op_{\max}j} \wedge \pi^0_{g_i.op_{\min}l}) \vee (\pi^0_{g_i.op_{\max}l} \wedge \pi^0_{g_i.op_{\min}j}) \text{ for } i \in [1, n]$$

- **Bounded SWAP selection.** For every step $t > 0$, adjacent edges can't be selected for SWAP at the same time to avoid simultaneous SWAPs on the same qubit, i.e., for any two edges d_k and $d_{k'}$ sharing a common vertex,

$$\neg c^t_k \vee \neg c^t_{k'} \text{ for } t \in [1, T], k, k' \in [1, K]$$

  The total number of SWAPs over T steps is at most S, i.e.,

$$\sum_{t=1}^{T} \sum_{k=1}^{K} c^t_k \leq S \tag{2}$$

  where the cardinality constraint (Equation 2) can be encoded into clauses using the Totalizer encoding detailed in Section 2.3.2.
- **Bijective layout mapping.** The layout mapping π^t_{ij} after each step t is required to be injective, i.e.,

$$\begin{aligned} &\text{AMO}(\{\pi^t_{ij}\}^P_{j=1}) \text{ for } i \in [1, V], \text{ and } t \in [0, T] \\ &\text{AMO}(\{\pi^t_{ij}\}^V_{i=1}) \text{ for } j \in [1, P], \text{ and } t \in [0, T] \end{aligned}$$

- **Layout mapping update.** After each step $t > 0$, π_{ij}^{t-1} will be updated to π_{ij}^{t}. If a SWAP in step t is performed on the k-th edge d_k in E (w.l.o.g., we assume d_k stores edge e_{ij}, i.e., the vertices i and j are endpoints of edge k.), then we have the following: (1) if an algorithmic qubit m is mapped to physical qubit i after step $t-1$ (i.e., $\pi_{mi}^{t-1} = \text{True}$), then $\pi_{mj}^{t} = \text{True}$:

$$\begin{aligned} &(c_k^t \wedge \pi_{mi}^{t-1}) \rightarrow \pi_{mj}^t \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T] \\ \Rightarrow\ &\neg c_k^t \vee \neg \pi_{mi}^{t-1} \vee \pi_{mj}^t \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T] \end{aligned}$$

  (2) similarly, if an algorithmic qubit m is mapped to physical qubit j after step $t-1$, then π_{mi}^{t} =True,

$$\neg c_k^t \vee \neg \pi_{mj}^{t-1} \vee \pi_{mi}^t \quad \text{for } m \in [1,V], k \in [1,K], t \in [1,T]$$

  (3) if an algorithmic qubit m is not mapped to i, j, then the qubit mapping after step $t-1$ and after step t are the same,

$$\neg c_k^t \vee \neg \pi_{ml}^{t-1} \vee \pi_{ml}^t \quad \text{for } m \in [1,V], l \in [1,P], l \neq i, l \neq j, k \in [1,K], t \in [1,T]$$

4.5 Complexity Analysis

Theorem 1 states the asymptotic number of variables and clauses introduced by our encoding. The encoding size increases approximately linearly with the number of transition steps.

▶ **Theorem 1.** *Given a quantum circuit of n gates, a quantum device of k edges in its connectivity graph, and t transition steps with s SWAP gates, the* SATmapper *encoding requires $\mathcal{O}(tk(n + \log tk))$ variables and $\mathcal{O}(tk(k + s + n^2))$ clauses.*

Proof. Assume the quantum circuit has v algorithmic qubits and the device has p physical qubits. We first consider the number of variables. We have $\mathcal{O}(tn + tvp + tk)$ original variables introduced in Section 4.3. The gate schedule continuation introduces $\mathcal{O}(tnk)$ auxiliary variables for resolving the big disjunction. The bounded SWAP selection introduces $\mathcal{O}(tk \log tk)$ auxiliary variables for the cardinality constraint. The bijective layout mapping introduces $\mathcal{O}(tpv)$ auxiliary variables for AMO constraints. As a result, we have $\mathcal{O}(tvp + tnk + tk \log tk) \subseteq \mathcal{O}(tk(n + \log tk))$ variables in total, assuming, w.l.o.g., $v \in \mathcal{O}(n)$ and $p \in \mathcal{O}(k)$.

For the number of clauses, the gate schedule initialization requires $\mathcal{O}(tn)$ clauses and the gate dependency constraints produces $\mathcal{O}(tn^2)$ clauses. The gate schedule continuation has $\mathcal{O}(tnk)$ original clauses and $\mathcal{O}(tnk)$ auxiliary clauses introduced to resolve the big disjunction. The bounded SWAP selection requires $\mathcal{O}(tk^2)$ clauses for non-conflict selection and $\mathcal{O}(tks)$ auxiliary clauses for encoding the cardinality constraint. Finally, the bijective layout mapping introduces $\mathcal{O}(tvp)$ clauses to encode AMO constraints, and the layout mapping update produces $\mathcal{O}(tkv^2)$ clauses. In total, we have $\mathcal{O}(tn^2 + tnk + tk^2 + tks + tvp + tkv^2) \subseteq \mathcal{O}(tk(k + s + n^2))$ clauses assuming $v \in \mathcal{O}(n)$ and $p \in \mathcal{O}(k)$. ◀

5 Experimental Evaluation

To evaluate our approach, we have implemented a prototype of SATmapper and compared it with the state-of-the-art solver-based method, TB-OLSQ2, and the best heuristic approach,TKET. We have conducted a comprehensive evaluation on seventy-eight instances ranging over two quantum computer devices, OQC Lucy [35] and Rigetti Aspen M-3 [36], and three quantum algorithms including QAOA ansatz for k-regular graphs, Quantum Fourier Transform (QFT), and Quantum Volume (QV) circuits, with various numbers of algorithmic qubits. All experiments were conducted on Amazon EC-2 instances c6a.48xlarge featuring 192 CPUs.

We used a time limit of 1800 seconds per instance. We set 16 CPUs and recorded the wall-clock time for parallel solvers. We set the step number T to be the same as the output of TKET. The runtime of TKET is less than one second on the given instances. We don't include this time since it is negligible compared to the solving time. The Rigetti Aspen M-3 and OQC Lucy devices contain 32 and 8 physical qubits, respectively, and the number of algorithmic qubits to be mapped cannot be more than the available number of physical qubits. We define the following three metrics to measure the performance of SWAP count optimization for SATmapper and TB-OLSQ2. Lower is better for all metrics and both SATmapper and TB-OLSQ2 start with an initial upper bound obtained by TKET.

- **Failure Ratio.** The ratio of instances where no improvement against TKET is observed.
- **SWAP Ratio.** The ratio of the best SWAP count to that of TKET on average.
- **Median Runtime.** The median runtime across all instances.

Particularly, we aim to address the following questions:

RQ1. Is SATmapper able to outperform TB-OLSQ2?
RQ2. What is the best underlying solver for SATmapper?

Summary. SATmapper outperformed TB-OLSQ2 on all three metrics, achieving a reduction of 57 percent for failure ratio and six percent for SWAP ratio, and lowering the median runtime from 237 to 9 seconds. Additionally, IncParKissat is the best-performing underlying solver for SATmapper by combining incremental and parallel solving.

5.1 RQ1. SATmapper vs. TB-OLSQ2

Table 1 summarizes the comparison between TB-OLSQ2 and SATmapper (equipped with IncParKissat) through the three metrics and note that the lower is better for all metrics. The first row presents the failure ratio, where TB-OLSQ2 had no improvement on 81% instances while SATmapper only failed on 24% instances achieving a reduction of 57 percent. In the second row, TB-OLSQ2 attained a SWAP ratio of 0.80 on average while SATmapper lowered the ratio to 0.74 by an improvement of six percent. Finally, the median runtime of TB-OLSQ2 is 237 seconds across all instances. As a comparison, SATmapper median runtime is only 9 seconds, which achieves a 26-fold reduction compared to TB-OLSQ2.

Table 1 Performance comparison between TB-OLSQ2 and SATmapper.

Metric	TB-OLSQ2	SATmapper
Failure Ratio	0.81	**0.24**
SWAP Ratio	0.80	**0.74**
Median Runtime/s	237	**9**

Table 2 presents a detailed comparison on a subset of instances. The first column gives the instance name in a format that lists device name, algorithm name, and number of algorithmic qubits. The next three columns compare the number of SWAPs optimized by TKET, TB-OLSQ2, and SATmapper. On easy instances with fewer than ten algorithmic qubits, both TB-OLSQ2 and SATmapper can improve the results against TKET and SATmapper can

Table 2 SWAP optimization comparison on a subset of instances.

	SWAP Count			Runtime/s	
Instance	TKET	TB-OLSQ2	SATmapper	TB-OLSQ2	SATmapper
Aspen-qaoa-3reg-n-8	5	3	3	1.97	3.32
Aspen-qaoa-3reg-n-10	6	4	4	1.67	2.37
Aspen-qaoa-3reg-n-12	19	8	7	116.85	1732.92
Aspen-qaoa-3reg-n-14	11	8	8	192.78	5.33
Aspen-qaoa-3reg-n-16	17	12	11	925.10	414.73
Aspen-qft-n-8	18	13	12	201.95	45.80
Aspen-qft-n-9	24	19	16	793.87	447.15
Aspen-qft-n-10	37	Failed	24	Failed	609.81
Aspen-qft-n-11	58	Failed	34	Failed	1335.53
Aspen-qft-n-12	55	Failed	40	Failed	1651.05
Aspen-qv-n-8	11	10	8	89.19	18.43
Aspen-qv-n-10	47	Failed	27	Failed	1640.08
Aspen-qv-n-12	56	Failed	50	Failed	513.64
OQC-qaoa-3reg-n-8	7	7	7	0.17	1.12
OQC-qft-n-8	28	23	23	14.15	4.44
OQC-qv-n-8	17	13	13	3.03	5.59

find lower SWAP counts than TB-OLSQ2. On hard instances of at least ten algorithmic qubits, TB-OLSQ2 failed to produce any better result for the QFT and QV instances while SATmapper still improved the results of TKET by around 15 SWAPs.

The last two columns of Table 2 compare the runtime. SATmapper is slightly slower on easy instances that TB-OLSQ2 could solve within a few seconds because of a constant overhead, but SATmapper is consistently faster than TB-OLSQ2 on harder instances. We still observe that SATmapper spent a significantly larger time than TB-OLSQ2 on some instances, for example, Aspen-qaoa-3reg-n-12. This is because SATmapper tried to find a lower SWAP count than TB-OLSQ2, which considerably increases the difficulty. If SATmapper stopped at eight SWAPs on this instance, the total runtime would be twenty seconds only, which is lower than the runtime of TB-OLSQ2, but the effort to find a mapping of seven SWAPs took the remaining 1712 seconds. The example also revealed the drastically increased difficulty when lowering the SWAP count. It's worth highlighting that SATmapper even used less time to produce a lower SWAP count than TB-OLSQ2 on many instances such as Aspen-qaoa-3reg-n-16, Aspen-qft-n-8, and Aspen-qv-n-8.

5.2 RQ2. Underlying Solver for SATmapper

Table 3 compares the performance of SATmapper when using different underlying SAT solvers. Specifically, we aim to benchmark the performance of incremental and parallel-solving techniques for SATmapper. Every column indicates SATmapper equipped with a particular solver. Kissat [9] is a state-of-the-art sequential SAT solver without incremental and parallel-solving techniques. CaDiCaL refers to using the CaDiCaL SAT solver [9] in incremental mode. Pbop is a non-incremental, clause-sharing parallel SAT solver based on Kissat. IncParKissat is the incremental and parallel SAT solver described in Section 3.3.2. Table 3 indicates that SATmapper with IncParKissat achieved the best performance on all three metrics. Particularly, both CaDiCaL and Pbop outperformed Kissat, which reveals the

Table 3 Performance comparison for SATmapper with different underlying solvers.

Metric	Kissat	CaDiCaL	Pbop	IncParKissat
Failure Ratio	0.27	0.26	0.25	**0.24**
SWAP Ratio	0.78	0.78	0.76	**0.74**
Median Runtime/s	183	137	18	**9**

individual benefits gained from incremental and parallel solving, respectively. Finally, the combination of incremental and parallel solving allows IncParKissat to benefit from the best of both worlds.

6 Conclusion

SWAP count optimization during quantum circuit compilation is critical to deploying quantum algorithms on current generation quantum devices. The existing solver-based method to address this problem does not scale well, but the fast heuristic approach tends to produce low-quality results. Our approach, based on SAT solving, outperforms the existing solver-based approach. It scales better and can produced smaller SWAP counts. It also produce higher-quality quantum circuits that the best heuristics methods. We implement the SWAP count optimization as a series of calls to a SAT solver. We introduced a novel SAT encoding, and developed an efficient implementation by combining incremental and parallel-solving techniques. A comprehensive evaluation on real-world quantum algorithms and devices demonstrates that our method is 26× faster than the existing solver-based approach and produces better results. Our method also improved on the heuristic approach on 76% of instances and achieved an average of 26% reduction in SWAP count.

References

1 Amira Abbas, Andris Ambainis, Brandon Augustino, Andreas Bärtschi, Harry Buhrman, Carleton Coffrin, Giorgio Cortiana, Vedran Dunjko, Daniel J Egger, Bruce G Elmegreen, et al. Quantum optimization: Potential, challenges, and the path forward. *arXiv preprint*, 2023. `arXiv:2312.02279`.

2 MD Sajid Anis, Héctor Abraham, R Agarwal AduOffei, Gabriele Agliardi, Merav Aharoni, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, M Amy, S Anagolum, et al. Qiskit: An open-source framework for quantum computing (2021). *SUPPLEMENTARY INFORMATION I. ALGORITHMS II. A RELAXATION BOUND (| E| 2+ 1 9 α| E| 2+ α) Remark*, 2021.

3 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Proc. of SAT*, 2009.

4 Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. An adaptive parallel sat solver. In *Proc. of CP*, 2016.

5 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proc. of IJCAI*, 2009.

6 Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In *Proc. of CP*, 2003.

7 Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2022.

8 Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 2017.

9 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, 2020.

10 Adam Bouland, Wim van Dam, Hamed Joorati, Iordanis Kerenidis, and Anupam Prakash. Prospects and challenges of quantum finance. *arXiv preprint*, 2020. `arXiv:2011.06492`.

11 Fernando G. S. L. Brandão, Amir Kalev, Tongyang Li, Cedric Yen-Yu Lin, Krysta M. Svore, and Xiaodi Wu. Quantum SDP Solvers: Large Speed-ups, Optimality, and Applications to Quantum Learning, 2019.

12 Markus Büttner and Jussi Rintanen. Satisfiability planning with constraints on the number of actions. In *Proc. of ICAPS*, 2005.

13 Yudong Cao, Jonathan Romero, Jonathan P. Olson, Matthias Degroote, Peter D. Johnson, Mária Kieferová, Ian D. Kivlichan, Tim Menke, Borja Peropadre, Nicolas P. D. Sawaya, Sukin Sim, Libor Veis, and Alán Aspuru-Guzik. Quantum chemistry in the age of quantum computing. *Chemical Reviews*, 2019.

14 Don Coppersmith. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067*, 2002.

15 Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proc. of TACAS*, 2008.

16 Richard P. Feynman. Simulating Physics with Computers. *International Journal of Theoretical Physics*, 1982.

17 Craig Gidney and Martin Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 2021.

18 Andrés Gómez, Álvaro Leitao, Alberto Manzano, Daniele Musso, María R. Nogueiras, Gustavo Ordóñez, and Carlos Vázquez. A Survey on Quantum Computational Finance for Derivatives Pricing and VaR. *Archives of Computational Methods in Engineering*, 2022.

19 Youssefa Hamadi, Saidb Jabbour, and Lakhdarb Sais. Manysat: a parallel sat solver issue title: Parallel sat solving. *Journal on Satisfiability, Boolean Modeling and Computation*, 2009.

20 Maximilian Heisinger, Mathias Fleury, and Armin Biere. Distributed cube and conquer with paracooba. In *Proc. of SAT*, 2020.

21 Dylan Herman, Cody Googin, Xiaoyuan Liu, Alexey Galda, Ilya Safro, Yue Sun, Marco Pistoia, and Yuri Alexeev. A survey of quantum computing for finance. *arXiv preprint*, 2022. `arXiv:2201.02773`.

22 Marijn J. H. Heule. Chinese remainder encoding for hamiltonian cycles. In *Proc. of SAT*, 2021.

23 Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Hardware and Software: Verification and Testing*, 2012.

24 Tomi H. Johnson, Stephen R. Clark, and Dieter Jaksch. What is a quantum simulator? *EPJ Quantum Technology*, 2014.

25 Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat: A partial max-sat solver. *J. Satisf. Boolean Model. Comput.*, 2012.

26 Gushu Li, Yufei Ding, and Yuan Xie. Tackling the qubit mapping problem for nisq-era quantum devices. In *Proc. of ASPLOS*, 2019.

27 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In *Proc. of SAT*, 2016.

28 Wan-Hsuan Lin, Jason Kimko, Bochen Tan, Nikolaj Bjørner, and Jason Cong. Scalable optimal layout synthesis for nisq quantum processors. In *Proc. of DAC*, 2023.

29 J.P. Marques Silva and K.A. Sakallah. Conflict analysis in search algorithms for satisfiability. In *Proc. of ICTAI*, 1996.

30 Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for maxsat. In *Proc. of CP*, 2014.

31 A. Molavi, A. Xu, M. Diges, L. Pick, S. Tannu, and A. Albarghouthi. Qubit mapping and routing via maxsat. In *Proc. of MICRO*, 2022.

32 Michele Mosca, Joao Marcos Vensi Basso, and Sebastian R. Verschoor. On speeding up factoring with quantum SAT solvers. *Scientific Reports*, 2020.

33 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010.

34 Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 2019.

35 Oxford quantum circuits. `https://aws.amazon.com/braket/quantum-computers/oqc/`. Accessed: 2024-03-16.

36 Rigetti computing. `https://www.rigetti.com/`. Accessed: 2024-02-20.

37 Dominik Schreiber and Peter Sanders. Scalable sat solving in the cloud. In *Proc. of SAT*, 2021.

38 Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. An introduction to quantum machine learning. *Contemporary Physics*, 2015.

39 Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 1997.

40 P.W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. of FOCS*, 1994.

41 Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t$|$ket$\rangle$: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 2020.

42 Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proc. of SAT*, 2009.

43 Bochen Tan and Jason Cong. Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers*, 2020.

44 Bochen Tan and Jason Cong. Optimal qubit mapping with simultaneous gate absorption. In *Proc. of ICCAD*, 2021.

45 HANTAO ZHANG, MARIA PAOLA BONACINA, and JIEH HSIANG. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 1996.

Anytime Approximate Formal Feature Attribution

Jinqiang Yu
Department of Data Science and AI, Monash University, Melbourne, Australia
Australian Research Council OPTIMA ITTC, Melbourne, Australia

Graham Farr
Department of Data Science and AI, Monash University, Melbourne, Australia

Alexey Ignatiev
Department of Data Science and AI, Monash University, Melbourne, Australia

Peter J. Stuckey
Department of Data Science and AI, Monash University, Melbourne, Australia
Australian Research Council OPTIMA ITTC, Melbourne, Australia

Abstract

Widespread use of artificial intelligence (AI) algorithms and machine learning (ML) models on the one hand and a number of crucial issues pertaining to them warrant the need for explainable artificial intelligence (XAI). A key explainability question is: given this decision was made, what are the input features which contributed to the decision? Although a range of XAI approaches exist to tackle this problem, most of them have significant limitations. Heuristic XAI approaches suffer from the lack of quality guarantees, and often try to approximate Shapley values, which is not the same as explaining which features contribute to a decision. A recent alternative is so-called formal feature attribution (FFA), which defines feature importance as the fraction of formal abductive explanations (AXp's) containing the given feature. This measures feature importance from the view of formally reasoning about the model's behavior. Namely, given a system of constraints logically representing the ML model of interest, computing an AXp requires finding a minimal unsatisfiable subset (MUS) of the system. It is challenging to compute FFA using its definition because that involves counting over all AXp's (equivalently, counting over MUSes), although one can approximate it. Based on these results, this paper makes several contributions. First, it gives compelling evidence that computing FFA is intractable, even if the set of contrastive formal explanations (CXp's), which correspond to minimal correction subsets (MCSes) of the logical system, is provided, by proving that the problem is #P-hard. Second, by using the duality between MUSes and MCSes, it proposes an efficient heuristic to switch from MCS enumeration to MUS enumeration on-the-fly resulting in an adaptive explanation enumeration algorithm effectively approximating FFA in an anytime fashion. Finally, experimental results obtained on a range of widely used datasets demonstrate the effectiveness of the proposed FFA approximation approach in terms of the error of FFA approximation as well as the number of explanations computed and their diversity given a fixed time limit.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Machine learning

Keywords and phrases Explainable AI, Formal Feature Attribution, Minimal Unsatisfiable Subsets, MUS Enumeration

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.30

Supplementary Material *Software (Source Code)*: `https://github.com/jinqiang-yu/Anytime-FFA`
archived at `swh:1:dir:61cea620ac438837adf74f0e128ef8f3bfaeea29`

Funding This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

1 Introduction

The rise of the use of artificial intelligence (AI) and machine learning (ML) methods to help interpret data and make decisions has exposed a keen need for these algorithms to be able to explain their decisions/judgements. Lack of explanation of opaque and complex models leads to lack of trust, and allows the models to encapsulate unfairness, discrimination and other unwanted properties learnt from the data or through training.

For a classification problem, a key explainability question is: "given a decision was made (a class was imputed to some data instance), what are the features that contributed to the decision?". A more complex question is: "given the decision was made, how important was each feature in making that decision?". There are many heuristic approaches to answering this question, mostly based on sampling around the instance [49], and attempting to approximate Shapley values [32]. But there is strong evidence that Shapley values do not really compute the importance of a feature to a decision [16, 35].

By building on techniques for handling over-constrained systems and minimal unsatisfiability [2, 31, 3, 34, 29], *formal* approaches to explainability (formal explainable AI, FXAI) are able to compute formal *abductive explanations* (AXp's) for a decision, that is a minimal set of features which are enough to ensure the same decision will be made [51, 21]. Namely, an abductive explanation can be associated with a minimal unsatisfiable subset (MUS) of a set of clauses logically representing the decision function of an ML classifier [20]. FXAI approaches can also compute formal *contrastive explanations* (CXp's), that is a minimal set of features, which must change in order to change the decision [39, 20]. Similarly to the case of AXps's, these can be associated with minimal correction subsets (MCSes) of a logical representation of the decision function [20]. Hence a wealth of algorithms originating from minimal unsatisfiability and over-constrained systems [2, 31, 3, 45, 28, 26, 34, 29] are directly applicable for the computation and enumeration of AXp's and CXp's [20, 36]. Here, enumeration of formal explanations builds on the use of the minimal hitting set duality between AXp's and CXp's [20] and the application of the well-known MARCO algorithm originally proposed for implicit hitting set based enumeration of MUSes of unsatisfiable CNF formulas [45, 27, 29]. Until recently there was no formal approach to ascribing importance to features.

A recent and attractive approach to formal feature attribution, called FFA [56], is simple. Compute all the abductive explanations for a decision, then the importance of a feature for the decision is simply the proportion of abductive explanations in which it appears. FFA is crisply defined, and easy to understand, but it is challenging to compute, as deciding if a feature has a non-zero attribution is at least as hard as deciding feature relevancy [15, 56].

Yu et al. [56] show that FFA can be efficiently computed by making use of the hitting set duality between AXp's and CXp's. By trying to enumerate CXp's, a side effect of the algorithm is to discover AXp's. In fact, the algorithm will usually find many AXp's before finding the first CXp. The AXp's are guaranteed to be diverse, since they need to be broad in scope to ensure that the CXp is large enough to hit all AXp's that apply to the decision.

Using AXp's collected as a side effect of CXp enumeration is effective at the start of the enumeration. But as we find more and more AXp's as side effects we eventually get to a point where many more CXp's are generated than AXp's. Experimentation shows that if we wish to enumerate all AXp's then indeed we should not rely on the side effect behavior, but simply enumerate AXp's directly. This leads to a quandary: to get fast accurate approximations of FFA we wish to enumerate CXp's and generate AXp's as a side effect. But to compute the final correct FFA we wish to compute all AXp's, and we are better off directly enumerating AXp's.

In this paper, we develop an *anytime* approach to computing approximate FFA, by starting with CXp enumeration, and then dynamically switching to AXp enumeration when the rate of AXp discovery by CXp enumeration drops. In doing so, we are able to quickly get accurate approximations, but also arrive to the full set of AXp's quicker than pure CXp enumeration. As direct CXp enumeration is feasible to do without the need to resort to the hitting set duality [36], one may want to estimate FFA by first enumerating CXp's. The second contribution of this paper is to investigate this alternative approach and to show that even if a(n) (in)complete set of CXp's is given, determining FFA is computationally expensive being #P-hard even if all CXp's are of size two.

The paper is organized as follows. The next section introduces the notation used throughout the paper. The main results of the paper are given in Section 3, which (1) theoretically argues that exact FFA computation is computationally hard and (2) it shows how to efficiently approximate FFA during the entire explanation enumeration process, which is done by switching from CXp enumeration to AXp enumeration on the fly. Section 4 provides experimental evidence that the proposed switching scheme is beneficial in practice as it helps us get to better quality approximations of FFA if compared to the standard setups of MARCO. Finally, Section 5 concludes the paper.

2 Preliminaries

Here we introduce the required propositional satisfiability (SAT) related notation as well as background on formal explainable AI in order to define formal feature attribution (FFA).

2.1 Satisfiability and Minimal Unsatisfiability

We assume standard definitions for propositional satisfiability (SAT) and maximum satisfiability (MaxSAT) solving [5]. A propositional formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A *clause* is a disjunction of literals. A *literal* is either a Boolean variable or its negation. Whenever convenient, clauses are treated as sets of literals while CNF formulas are treated as *sets of clauses.* A truth assignment maps each variable of a formula to a value from $\{0, 1\}$. Given a truth assignment, a clause is said to be satisfied if at least one of its literals is assigned value 1; otherwise, it is falsified by the assignment. A formula is satisfied if all of its clauses are satisfied; otherwise, it is falsified. If there exists no assignment that satisfies a CNF formula, then the formula is *unsatisfiable.*

In the context of unsatisfiable formulas, the maximum satisfiability (MaxSAT) problem is to find a truth assignment that maximizes the number of satisfied clauses. While a number of variants of MaxSAT exist [5, Chapters 23 and 24], hereinafter, we are interested in Partial Unweighted MaxSAT, which can be formulated as follows. A formula ϕ is represented as a conjunction of *hard* clauses $\mathcal{H}$, which must be satisfied, and *soft* clauses $\mathcal{S}$, which represent a preference to satisfy those clauses, i.e. $\phi = \mathcal{H} \wedge \mathcal{S}$ (or $\phi = \mathcal{H} \cup \mathcal{S}$ in the set theory notation). The Partial Unweighted MaxSAT problem consists in finding an assignment that satisfies all the hard clauses and maximizes the total number of satisfied soft clauses. In the analysis of an unsatisfiable formula ϕ, one is also often interested in identifying minimal unsatisfiable subsets (MUSes) and minimal correction subsets (MCSes) of ϕ, which can be defined as follows[1].

[1] The problems we are tackling with these formalisms in this paper belong to decidable fragments of first-order logic. While the definitions provided here are given for the propositional case, their extension to the first-order case is straightforward.

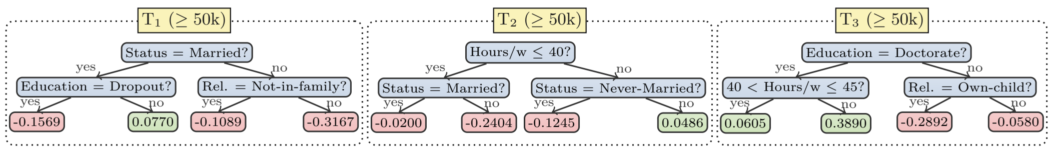


Figure 1 Example boosted tree model [8] trained on the well-known *adult* classification dataset.

▶ **Definition 1** (Minimal Unsatisfiable Subset (MUS)). *Let $\phi = \mathcal{H} \cup \mathcal{S}$ denote an unsatisfiable set of clauses, i.e. $\phi \vDash \bot$. A subset of clauses $\mu \subseteq \mathcal{S}$ is a* Minimal Unsatisfiable Subset *(MUS) iff $\mathcal{H} \cup \mu \vDash \bot$ and $\forall \mu' \subsetneq \mu$ it holds that $\mathcal{H} \cup \mu' \nvDash \bot$.*

Informally, an MUS can be seen as a minimal explanation of unsatisfiability for an unsatisfiable formula ϕ as it provides the minimal information that needs to be added to the hard clauses $\mathcal{H}$ to obtain unsatisfiability. Alternatively, one may be interested in *correcting* the formula by removing some of the clauses in $\mathcal{S}$ to achieve satisfiability.

▶ **Definition 2** (Minimal Correction Subset (MCS)). *Let $\phi = \mathcal{H} \cup \mathcal{S}$ denote an unsatisfiable set of clauses, i.e. $\phi \vDash \bot$. A subset of clauses $\sigma \subseteq \mathcal{S}$ is a* Minimal Correction Subset *(MCS) iff $\mathcal{H} \cup \mathcal{S} \setminus \sigma \nvDash \bot$ and $\forall \sigma' \subsetneq \sigma$ it holds that $\mathcal{H} \cup \mathcal{S} \setminus \sigma' \vDash \bot$.*

Informally, an MCS can be seen as a minimal way to "correct" unsatisfiability of an unsatisfiable formula ϕ. A fundamental result in reasoning about unsatisfiable CNF formulas is the minimal hitting set (MHS) duality relationship between MUSes and MCSes [48, 6]. That is if the sets of all MUSes and MCSes of formula ϕ are denoted as $\mathbb{U}_\phi$ and $\mathbb{C}_\phi$ then $\mathbb{U}_\phi = \text{MHS}(\mathbb{C}_\phi)$ and $\mathbb{C}_\phi = \text{MHS}(\mathbb{U}_\phi)$ where MHS(S) returns the minimal hitting sets of S, that is the minimal sets that share an element with each subset in S. More formally, $\text{HS}(S) = \{t \subseteq (\cup S) \mid \forall s \in S,\ t \cap s \neq \emptyset\}$ and $\text{mins}(S) = \{s \in S \mid \forall t \subsetneq s,\ t \notin S\}$ returns the subset-minimal elements of a set of sets, and $\text{MHS}(S) = \text{mins}(\text{HS}(S))$. This result has been extensively used in the development of algorithms for MUSes and MCSes [2, 31, 29], and also applied in a number of different settings. Recent years have witnessed the emergence of a large number of novel algorithms for the extraction and enumeration of MUSes and MCSes [38, 1, 29, 37, 46, 13, 43, 4].

2.2 Classification Problems

We assume classification problems classify data instances into classes $\mathcal{K}$ where $|\mathcal{K}| = k \geq 2$. We are given a set of m features $\mathcal{F}$, where the value of feature $i \in \mathcal{F}$ comes from a domain $\mathbb{D}_i$, which may be Boolean, (bounded) integer or (bounded) real. The *complete feature space* is defined by $\mathbb{F} \triangleq \prod_{i=1}^{m} \mathbb{D}_i$.

A *data point* in feature space is denoted $\mathbf{v} = (v_1, \ldots, v_m)$ where $v_i \in \mathbb{D}_i, 1 \leq i \leq m$. An *instance* of the classification problem is a pair of feature vector and its corresponding class, i.e. $(\mathbf{v}, c)$, where $\mathbf{v} \in \mathbb{F}$ and $c \in \mathcal{K}$.

We use the notation $\mathbf{x} = (x_1, \ldots, x_m)$ to represent an arbitrary point in feature space, where each x_i will take a value from $\mathbb{D}_i$.

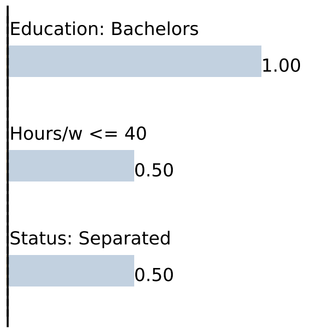
A *classifier* is a total function from feature space to class: $\kappa : \mathbb{F} \to \mathcal{K}$. Many approaches exist to define classifiers including decision sets [9, 25], decision lists [50], decision trees [18], random forests [11], boosted trees [8], and neural nets [42, 17].

$\mathcal{X}_1 = \{$ **Education, Hours/w** $\}$	
IF	Education = Bachelors
AND	Hours/w ≤ 40
THEN	Target $<50k$

(a) AXp's $\mathcal{X}_1$.

$\mathcal{X}_2 = \{$ **Education, Status** $\}$	
IF	Education = Bachelors
AND	Status = Separated
THEN	Target $<50k$

(b) AXp's $\mathcal{X}_2$.



(c) FFA.

Figure 2 Examples of both AXp's (no more AXp's exist) followed by FFA for the instance **v** shown in Example 3 as well as formal feature attribution (FFA).

▶ **Example 3.** Figure 1 shows a boosted tree (BT) model trained with the use of XGBoost [8] for a simplified version of the *adult* dataset [23]. BT models comprise an ensemble of decision trees; given an instance to classify, each decision tree in a BT model contributes a numeric weight to a particular class and the class with the largest total weight is selected as the model's prediction. For a data instance $\mathbf{v} = \{\text{Education} = \text{Bachelors, Status} = \text{Separated}, \text{Occupation} = \text{Sales, Relationship} = \text{Not-in-family}, \text{Sex} = \text{Male, Hours/w} \leq 40\}$, the model predicts $<50k$ because the sum of the weights in the 3 trees for this instance equals $-0.4073 = (-0.1089 - 0.2404 - 0.0580) < 0$.

2.3 Formal Explainability

Given a data point $\mathbf{v}$, classifier κ classifies it as class $\kappa(\mathbf{v})$. A *post-hoc explanation* of the behavior of κ on data point $\mathbf{v}$ tries to explain the behavior of κ on this instance. We consider two forms of formal explanation answering *why* and *why not* (or *how*) questions.

An *abductive explanation* (AXp) is a minimal set of features $\mathcal{X}$ such that any data point sharing the same feature values with $\mathbf{v}$ on these features is guaranteed to be assigned the same class $c = \kappa(\mathbf{v})$ [51, 21]. Formally, $\mathcal{X}$ is a subset-minimal set of features such that:

$$\forall(\mathbf{x} \in \mathbb{F}).\ \left[\bigwedge\nolimits_{i \in \mathcal{X}} (x_i = v_i)\right] \rightarrow (\kappa(\mathbf{x}) = c) \tag{1}$$

▶ **Example 4.** In the context of Figure 1, the two AXp's for the instance $\mathbf{v}$ are shown in Figure 2a and Figure 2b. AXp $\mathcal{X}_1$ indicates that specifying *Education* = *Bachelors* and $\mathit{Hours/w} \leq 40$ guarantees that any compatible instance is classified as < 50k independent of the values of other features, e.g. *Status* and *Relationship*, since the maximal sum of weights is $0.0770 - 0.0200 - 0.0580 = -0.0010 < 0$ as long as the feature values above are used. Observe that another AXp $\mathcal{X}_1$ for $\mathbf{v}$ is $\{\mathit{Education}, \mathit{Status}\}$, i.e. the model is guaranteed to predict < 50k for any instance in the feature space where features *Education* and *Status* have values *Bachelors* and *Separated*, respectively. Note that no more AXp's exist for instance $\mathbf{v}$. Since both of the two AXp's for $\mathbf{v}$ consist of two features, it is difficult to judge which one is better without a formal feature importance assessment.

▶ **Example 5.** Consider again the ensemble shown in Figure 1. It contains only features *Status*, *Education*, *Relationship*, and *Hours/w*, which can be denoted by integer variables s, e, r, and h, respectively. Note that all the other features of this dataset do not take part in the classification process and can be ignored. Let us map *Status* values *married* and *never-married* to value 1 and 2 of s while value 0 represents all other values. Similarly, we can assign values *dropout*, *doctorate* and *any other value* of feature *Education* to values 1, 2, and 0 of variable e; values *not-in-family*, *own-child*, and *any other value* of feature *Relationship* to values 1, 2, and 0 of variable r. This way $\mathbb{D}_s = \mathbb{D}_e = \mathbb{D}_r = \{0, 1, 2\}$. Finally, according to the tree ensemble, $\mathbb{D}_h = \mathbb{Z}$. As a result and assuming the values assigned by the trees are represented by variables $t_i \in \mathbb{R}$, the classification process for instance $\mathbf{v}$ in Example 3 (predicted as $< 50k$) can be expressed as the following set of *hard* constraints, which are simple to represent in clausal form:

$$\mathcal{H} = \begin{cases} t_1 = -0.1569 & \leftrightarrow \quad (s = 1 \wedge e = 1) \\ t_1 = -0.0770 & \leftrightarrow \quad (s = 1 \wedge (e = 0 \vee e = 2)) \\ t_1 = -0.1089 & \leftrightarrow \quad ((s = 0 \vee s = 2) \wedge r = 1) \\ \quad\dots & \\ t_2 = -0.2404 & \leftrightarrow \quad (h \leq 40 \wedge (s = 0 \vee s = 2)) \\ \quad\dots & \\ t_3 = -0.2892 & \leftrightarrow \quad ((e = 0 \vee e = 1) \wedge r = 2) \\ t_3 = -0.0580 & \leftrightarrow \quad ((e = 0 \vee e = 1) \wedge (r = 0 \vee r = 1)) \\ t_1 + t_2 + t_3 < 0 & \end{cases}$$

Observe that instance $\mathbf{v}$ can be specified as a set of *soft* unit clauses $\mathcal{S} = \{(e = 0), (s = 0), (r = 1), (h \leq 40)\}$. Observe that formula $\mathcal{H} \wedge \mathcal{S}$ is unsatisfiable having two MUSes $\{(e = 0), (h \leq 40)\}$ and $\{(e = 0), (s = 0)\}$, which correspond to the two AXp's shown in Example 4.

A dual concept of *contrastive explanations* (CXp's) helps us understand *how* to reach another prediction [39, 20, 36]. A *contrastive explanation* (CXp) for the classification of data point $\mathbf{v}$ with class $c = \kappa(\mathbf{v})$ is a minimal set of features that must change so that κ can return a different class. Formally, a CXp is a subset minimal set of features $\mathcal{Y}$ such that

$$\exists(\mathbf{x} \in \mathbb{F}).\ \left[\bigwedge\nolimits_{i \notin \mathcal{Y}} (x_i = v_i)\right] \wedge (\kappa(\mathbf{x}) \neq c) \tag{2}$$

It is known [20] that formal abductive and contrastive explanations for ML predictions are related with the concepts of MUSes and MCSes (defined earlier) of an *unsatisfiable* formula encoding the ML classification process $\kappa(\mathbf{v}) = c$, namely if one represents $[\kappa(\mathbf{x}) \neq c]$ as hard clauses and $[\bigwedge_{i=1}^{m} (x_i = v_i)]$ as soft clauses. For this reason, the set $\mathbb{A}$ of all AXp's $\mathcal{X}$ explaining classification $\kappa(\mathbf{v}) = c$ and the set $\mathbb{C}$ of all CXp's $\mathcal{Y}$ explaining the same classification enjoy a *minimal hitting set duality* [20], similarly to MUSes and MCSes. That is $\mathbb{A} = \text{MHS}(\mathbb{C})$ and is $\mathbb{C} = \text{MHS}(\mathbb{A})$. This property can be made use of when computing or enumerating AXp's and/or CXp's [20, 33, 36].

▶ Remark 6. Thanks to the relation between AXp's (resp., CXp's) for a given ML prediction on the one hand and MUSes (resp., MCSes) of formula encoding the decision process on the other hand, all the ideas and algorithms considered in this paper can be directly applied in any context where MUSes and MCSes are of use.

▶ **Example 7.** Consider the BT model and instance $\mathbf{v}$ in Example 2. Observe that $\mathcal{Y} = \{Education\}$ is a CXp for instance $\mathbf{v}$ since the prediction for this instance can be changed if feature *Education* is allowed to take another value, e.g. changing the value of feature *Education* to *Doctorate* triggers that the sum of the weights in the 3 trees becomes $-0.1089 - 0.2404 + 0.3890 = 0.0397 > 0$. By further examining the model and $\mathbf{v}$, more subsets of features can be identified as CXp's for $\mathbf{v}$. The complete set of CXp's for this instance is $\{\{Education\}, \{Status, Hours/w\}\}$, which minimally hits the set of AXp's shown in Example 4. Also observe that the set of CXp's corresponds to the set of MCSes of formula $\mathcal{H} \wedge \mathcal{S}$ shown in Example 5: $\{(e = 0)\}$ and $\{(h \leq 40), (s = 0)\}$.

2.4 Formal Feature Attribution

Given the definition of AXp's above, we can now illustrate the *formal feature attribution* (FFA) function by Yu *et al* [56]. Denoted as $\text{ffa}_\kappa(i, (\mathbf{v}, c))$, it returns for a classification $\kappa(\mathbf{v}) = c$ how important feature $i \in \mathcal{F}$ is in making this classification, defined as the proportion of AXp's for the classification $\mathbb{A}_\kappa(\mathbf{v}, c)$, which include feature i, i.e.

$$\text{ffa}_\kappa(i, (\mathbf{v}, c)) = \frac{|\{\mathcal{X} \mid \mathcal{X} \in \mathbb{A}_\kappa(\mathbf{v}, c), i \in \mathcal{X}\}|}{|\mathbb{A}_\kappa(\mathbf{v}, c)|} \tag{3}$$

▶ **Example 8.** Recall Example 4. As there are 2 AXp's for instance $\mathbf{v}$, namely $\{Education, Status\}$ and $\{Education, Hours/w\}$, the prediction can be attributed to the 3 features with non-zero FFA shown in Figure 2c. Namely, features *Education*, *Status*, and *Hours/w* have the attribution values of 1, 0.5, and 0.5, respectively.

2.5 Computing FFA

Inspired by the implicit hitting set [7] based algorithm eMUS/MARCO [45, 26, 19] for enumerating MUSes and MCSes of an unsatisfiable CNF formula, Yu *et al* [56] define an anytime algorithm for computing FFA shown in Algorithm 1. The algorithm collects AXp's $\mathbb{A}$ and CXp's $\mathbb{C}$. They are initialized to empty. While we still have resources, we generate a minimal hitting set $\mathcal{Y} \in \text{MHS}(\mathbb{A})$ of the current known AXp's $\mathbb{A}$ and not already in $\mathbb{C}$ with the call $\textsc{MinimalHS}(\mathbb{A}, \mathbb{C})$. If no (new) hitting set exists then we are finished and exit the loop. Otherwise we check if (2) holds in which case we add the candidate to the set of CXp's $\mathbb{C}$. Otherwise, we know that $\mathcal{F} \setminus \mathcal{Y}$ is a correct (non-minimal) abductive explanation, i.e. it satisfies (1). We use the call $\textsc{ExtractAXp}$ to minimize the resulting explanation, returning an AXp $\mathcal{X}$ which is added to the collection of AXp's $\mathbb{A}$. $\textsc{ExtractAXp}$ tries to remove features j from $\mathcal{F} \setminus \mathcal{Y}$ one by one while still satisfying (1). When resources are exhausted, the loop exits and we return the set of AXp's and CXp's currently discovered.

2.6 Graph-Related Notation

The paper uses some (undirected) graph-theoretic concepts. A graph is defined as a tuple, $G = (V, E)$, where V is a finite set of vertices and E is a finite set of unordered pairs of vertices. For simplicity, uv denotes an edge $\{u, v\}$ of E. Given a graph $G = (V, E)$, a *vertex cover* $X \subseteq V$ is such that for each $uv \in E$, $\{u, v\} \cap X \neq \emptyset$. A *minimal* vertex cover is a vertex cover that is minimal wrt. set inclusion.

2.7 The Complexity of Counting

The class #P consists of functions that count accepting computations of polynomial-time non-deterministic Turing machines [53]. A problem is *#P-hard* if every problem in #P is polynomial-time Turing reducible to it; if it also belongs to #P then it is *#P-complete.*

#P-hardness is usually regarded as stronger evidence of intractability than NP-hardness or indeed hardness for any level of the Polynomial Hierarchy.

Algorithm 1 Anytime Explanation Enumeration as defined by Yu *et al* [56].

```
procedure XpEnum($\kappa$, $\mathbf{v}$, $c$)
    $(\mathbb{A}, \mathbb{C}) \leftarrow (\emptyset, \emptyset)$
    while resources available do
        $\mathcal{Y} \leftarrow$ MinimalHS$(\mathbb{A}, \mathbb{C})$
        if $\mathcal{Y} = \bot$ then break
        if $\exists(\mathbf{x} \in \mathbb{F}).\left[\bigwedge_{i \notin \mathcal{Y}}(x_i = v_i)\right] \wedge (\kappa(\mathbf{x}) \neq c)$ then
            $\mathbb{C} \leftarrow \mathbb{C} \cup \{\mathcal{Y}\}$
        else
            $\mathcal{X} \leftarrow$ ExtractAXp$(\mathcal{F} \setminus \mathcal{Y}, \kappa, \mathbf{v}, c)$
            $\mathbb{A} \leftarrow \mathbb{A} \cup \{\mathcal{X}\}$
    return $\mathbb{A}$, $\mathbb{C}$
procedure ExtractAXp($\mathcal{X}$, $\kappa$, $\mathbf{v}$, $c$)
    for $j \in \mathcal{X}$ do
        if $\forall(\mathbf{x} \in \mathbb{F}).\left[\bigwedge_{i \in \mathcal{X} \setminus \{j\}}(x_i = v_i)\right] \rightarrow (\kappa(\mathbf{x}) = c)$ then
            $\mathcal{X} \leftarrow \mathcal{X} \setminus \{j\}$
    return $\mathcal{X}$
```

3 Approximate Formal Feature Attribution

Facing the need to compute (exact or approximate) FFA values, one may think of a possibility to first enumerate CXp's and then apply the minimal hitting set duality between AXp's and CXp's to determine FFA, without explicitly computing $\mathbb{A} = \text{MHS}(\mathbb{C})$. This looks plausible given that CXp enumeration can be done directly, without the need to enumerate AXp's [20]. However, as Section 3.1 argues, computing FFA given a set of CXp's turns out to be computationally difficult, being (roughly) at least as hard as counting the minimal hitting sets MHS($\mathbb{C}$). Hence, Section 3.2 approaches the problem from a different angle by efficient exploitation of the eMUS- or MARCO-like setup [45, 27, 29, 20] and making the algorithm *switch* from CXp enumeration to AXp enumeration on the fly.

3.1 Duality-Based Computation is Hard

We show that determining $\text{ffa}_\kappa(i, (\mathbf{v}, c))$ from $\mathbb{C}$ is #P-hard even when all CXp's have size two. In that special case, the CXp's may be treated as the edges of a graph, which we denote by $G(\mathcal{F}, \kappa, \mathbf{v}, c)$, with vertex set $\mathcal{F}$. The minimal hitting set duality between the CXp's and AXp's then implies that the AXp's $\mathcal{X} \in \text{MHS}(\mathbb{C})$ are precisely the minimal vertex covers of $G(\mathcal{F}, \kappa, \mathbf{v}, c)$. It is known that determining the number of minimal vertex covers in a graph is #P-complete (even for bipartite graphs); this is implicit in [47], as noted for example in [52, p. 400].

When all CXp's have size 2, the formal feature attribution $\text{ffa}_\kappa(i, (\mathbf{v}, c))$ is just the proportion of minimal vertex covers of $G(\mathcal{F}, \kappa, \mathbf{v}, c)$ that contain the vertex i, i.e. the vertex of $G(\mathcal{F}, \kappa, \mathbf{v}, c)$ that represents the feature $i \in \mathcal{F}$. To help express this in graph-theoretic language, write $\#\text{mvc}(G)$ for the number of minimal vertex covers of G. Write $\#\text{mvc}(G, v)$ and $\#\text{mvc}(G, \neg v)$ for the numbers of minimal vertex covers of G that *do* and *do not* contain vertex $v \in V(G)$, respectively. Define

$$\text{ffa}(G, v) := \frac{\#\text{mvc}(G, v)}{\#\text{mvc}(G)}. \tag{4}$$

Then

$$\text{ffa}_\kappa(i, (\mathbf{v}, c)) = \text{ffa}(G(\mathcal{F}, \kappa, \mathbf{v}, c), i).$$

Observe that $\#\text{mvc}(G) = \#\text{mvc}(G, v) + \#\text{mvc}(G, \neg v)$. Then we may rewrite (4) as

$$\text{ffa}(G, v) = \frac{\#\text{mvc}(G, v)}{\#\text{mvc}(G, v) + \#\text{mvc}(G, \neg v)}. \tag{5}$$

▶ **Theorem 9.** *Determining* ffa(G, v) *is #P-hard.*

Proof. We give a polynomial-time Turing reduction from the #P-complete problem of counting minimal vertex covers to the problem of determining ffa for a node in a graph.

Suppose we have an oracle that, when given a graph and a vertex, returns the ffa of the vertex in one time-step.

Let G be a graph for which we want to count the minimal vertex covers. Let v be a non-isolated vertex of G. (If none exists, the problem is trivial.) Put

$$\begin{aligned} x &= \#\text{mvc}(G, \neg v), \\ y &= \#\text{mvc}(G, v), \end{aligned}$$

so that $\#\text{mvc}(G) = x + y$. It is routine to show that $x, y > 0$. Initially, x and y are unknown. Our reduction will use an ffa-oracle to gain enough information to determine x and y. We will then obtain $\#\text{mvc}(G) = x + y$.

First, query the ffa-oracle with G and vertex v. It returns

$$p := \frac{y}{x + y},$$

by (5). We can then recover the ratio $x/y = p^{-1} - 1$.

Then we construct a new graph $G_v^{[2]}$ from G as follows. Take two disjoint copies G_1 and G_2 of G. Let v_1 be the copy of vertex v in G_1. For every $w \in V(G_2)$, add an edge $v_1 w$ between v_1 and w. We query the ffa-oracle with $G_v^{[2]}$ and vertex v_1. Let $q = \text{ffa}(G_v^{[2]}, v_1)$ be the value it returns.

If a minimal vertex cover X of $G_v^{[2]}$ contains v_1 then all the edges from v_1 to G_2 are covered. The restriction of X to G_1 must be a minimal vertex cover of G_1 that contains v_1, and the number of these is just $\#\text{mvc}(G, v) = y$. The restriction of X to G_2 must just be a vertex cover of G_2, without any further restriction, and the number of these is just $\#\text{mvc}(G) = x + y$. These two restrictions of X can be chosen independently to give all possibilities for X. So

$$\#\text{mvc}(G_v^{[2]}, v_1) = y(x + y).$$

If a minimal vertex cover X of $G_v^{[2]}$ does not contain v_1 then the edges $v_1 w$, $w \in V(G_2)$, are not covered by v_1. So each $w \in V(G_2)$ must be in X, which serves to cover not only those edges but also all edges in G_2. The restriction of X to G_1 must just be a vertex cover of G_1 that does not contain v_1, and there are $\#\text{mvc}(G, \neg v) = x$ of these. Again, the two restrictions of X are independent. So

$$\#\text{mvc}(G_v^{[2]}, \neg v_1) = x.$$

Therefore

$$q = \frac{y(x + y)}{x + y(x + y)},$$

by (5) (applied this time to $G_v^{[2]}$), so

$$x + y = \frac{x/y}{q^{-1} - 1} = \frac{p^{-1} - 1}{q^{-1} - 1}.$$

We can therefore compute $x + y$ from the values p and q returned by our two oracle calls. We therefore obtain $\#\text{mvc}(G)$. The entire computation is polynomial-time. ◀

▶ **Corollary 10.** *Determining* $\text{ffa}_\kappa(i, (\mathbf{v}, c))$ *from the set of CXp's is #P-hard, even if all CXp's have size 2.*

Unfortunately, Theorem 9 and Corollary 10 suggest that relying solely on the *direct* enumeration of CXp's in the fashion of the first phase of CAMUS-like algorithms [30, 31] when computing formal feature attribution does not make the problem simple. One will still need one to implicitly or explicitly enumerate AXp's to be able to compute FFA.

3.2 Switching from CXp to AXp Enumeration

As discussed in Section 2, [56] proposed to apply implicit hitting set enumeration for approximating FFA thanks to the duality between AXp's and CXp's. The approach builds on the use of the MARCO algorithm [45, 27, 29] in the anytime fashion, i.e. collects the sets of AXp's and CXp's and stops upon reaching a given resource limit. As MARCO can be set to target enumerating either AXp's or CXp's depending on user's preferences, the dual explanations will be collected by the algorithm as a *side effect.* Quite surprisingly, the findings of [56] show that for the purposes of *practical* FFA approximation it is beneficial to target CXp enumeration and get AXp's by duality. An explanation offered for this by [56] is that MARCO has to collect a large number of dual explanations before the minimal hitting sets it gets may realistically be the target explanations.

Our practical observations confirm the above. Also note that the AXp's enumerated by MARCO need to be *diverse* if we want to quickly get good FFA approximations. Due to the *incremental* operation of the minimal hitting set enumeration algorithms, this is hard to achieve if we *target* AXp enumeration. But if we aim for CXp's then we can extract diverse AXp's by duality, which helps us get reasonable FFA approximations quickly converging to the exact FFA values.

Nevertheless, our experiments with the setup of [56] suggest that AXp enumeration in fact tends to finish much earlier than CXp enumeration despite "losing" at the beginning. This makes one wonder what to opt for if good and quickly converging FFA approximation is required: AXp enumeration or CXp enumeration. On the one hand, the latter quickly gives a large set of diverse AXp's and good initial FFA approximations. On the other hand, complete AXp enumeration finishes much faster, i.e. exact FFA is better to compute by targeting AXp's.

Motivated by this, we propose to set up MARCO in a way that it starts with CXp enumeration and then seamlessly switches to AXp enumeration using two simple heuristic criteria. It should be first noted that to make efficient switching in the target explanations, we employ pure SAT-based hitting set enumerator, where an incremental SAT solver is called multiple times aiming for minimal or maximal models [12], depending on the phase. This allows us to keep all the explanations found so far without ever restarting the hitting set enumerator.

As we observe that AXp's are normally larger than CXp's, both criteria for switching the target build on the use of the average *size* of the last w AXp's and the last w CXp's enumerated in the most recent iterations of the MARCO algorithm. (Recall that our MARCO setup aims for subset-minimal explanations rather than cardinality-minimal explanations, i.e. neither target nor dual explanations being enumerated are cardinality-minimal.) This can be seen as inspecting "sliding windows" of size w for both AXp's and CXp's. In particular, assume that the sets of the last w AXp's and CXp's are denoted as $\mathbb{A}^w$ and $\mathbb{C}^w$, respectively.

First, switching can be done as soon as we observe that CXp's on average are *much* smaller than AXp's, i.e. when

$$\frac{\sum_{\mathcal{X}\in\mathbb{A}^w}|\mathcal{X}|}{\sum_{\mathcal{Y}\in\mathbb{C}^w}|\mathcal{Y}|} \geq \alpha, \tag{6}$$

where $\alpha \in \mathbb{R}$ is a predefined numeric parameter. The rationale for this heuristic is as follows. Recall that extraction of a subset-minimal *dual* explanation is done by ExtractDualXp() by means of deciding the validity of the corresponding predicate, either (1) or (2), while iteratively removing features from the feature set completementary to the candicate set, i.e. $\mathcal{F} \setminus \mu$ (see Section 2.5). As such, if the vast majority of CXp's are much smaller than their AXp counterparts, which implies that $|\mathcal{F} \setminus \mu| \gg |\mu|$, then extracting these dual AXp's from $\mathcal{F} \setminus \mu$ may be expensive as it leads to a large number of SAT oracle calls (namely $|\mathcal{F} \setminus \mu|$ calls) per dual AXp to extract. Hence, we prefer to switch the enumerator to the opposite phase such that ExtractDualXp() deals with a smaller number of decision oracle calls on average. Note that having small dual CXp's will also result in the lion's share of these oracle calls being *satisfiable*, i.e. potentially cheap.

Second, we can switch when the average CXp size "stabilizes". Here, let us denote a new CXp being just computed as $\mathcal{Y}_{\text{new}}$. Then the second criterion can be examined by checking if the following holds:

$$\left| |\mathcal{Y}_{\text{new}}| - \frac{\sum_{\mathcal{Y}\in\mathbb{C}^w}|\mathcal{Y}|}{w} \right| \leq \varepsilon, \tag{7}$$

with $\varepsilon \in \mathbb{R}$ being another numeric parameter. This condition is meant to signify the point when the set of dual AXp's we have already accumulated is diverse enough for all the CXp's to be of roughly equal size, which is crucial for good FFA approximations. Once we have reached a high level of FFA approximation, it makes sense to switch the target phase to AXp as it normally finishes exhaustive explanation enumeration earlier. Overall, the switching can be performed when either of the two conditions (6)–(7) is satisfied.

Algorithm 2 shows the pseudo-code of the adaptive explanation enumeration algorithm. Additionally to the classifier's representation κ, instance $\mathbf{v}$ to explain and its class c, it receives 3 numeric parameters: window size $w \in \mathbb{N}$ and switching-related constants $\alpha, \varepsilon \in \mathbb{R}$. The set of CXp's (resp. AXp's) is represented by $\mathbb{E}_0$ (resp. $\mathbb{E}_1$) while the target phase of the hitting set enumerator is denoted by $\rho \in \{0, 1\}$, i.e. at each iteration Algorithm 2 aims for $\mathbb{E}_\rho$ explanations. As initially $\rho = 0$, the algorithm targets CXp enumeration. Each of its iterations starts by computing a minimal hitting set μ of the set $\mathbb{E}_{1-\rho}$ subject to $\mathbb{E}_\rho$ (see line 5), i.e. we want μ to be a hitting set of $\mathbb{E}_{1-\rho}$ different from all the target explanations in $\mathbb{E}_\rho$ found so far. If no hitting set exists, the process stops as we have enumerated all target explanations. Otherwise, each new μ is checked for being a target explanation, which is done by invoking a reasoning oracle to decide the validity either of (1) if we target AXp's, or of (2) if we target CXp's. If the test is positive, the algorithm records the new explanation μ in $\mathbb{E}_\rho$. Otherwise, using the standard apparatus of formal explanations, it extracts a subset-minimal dual explanation ν from the complementary set $\mathcal{F} \setminus \mu$, which is then recorded in $\mathbb{E}_{1-\rho}$. Each iteration is additionally augmented with a check whether we should switch to the opposite phase $1 - \rho$ of the enumeration. This is done in line 12 by testing whether at least one of the conditions (6)–(7) is satisfied.

Algorithm 2 Adaptive Explanation Enumeration.

```
1: procedure AdaptiveXpEnum($\kappa$, $\mathbf{v}$, $c$, $w$, $\alpha$, $\varepsilon$)
2:     $(\mathbb{E}_0, \mathbb{E}_1) \leftarrow (\emptyset, \emptyset)$                         ▷ CXp's and AXp's to collect
3:     $\rho \leftarrow 0$                                            ▷ Target phase of enumerator, initially CXp
4:     while true do
5:         $\mu \leftarrow$ MinimalHS$(\mathbb{E}_{1-\rho}, \mathbb{E}_\rho, \rho)$
6:         if $\mu = \bot$ then break
7:         if IsTargetXp$(\mu, \kappa, \mathbf{v}, c)$ then
8:             $\mathbb{E}_\rho \leftarrow \mathbb{E}_\rho \cup \{\mu\}$                  ▷ Collect target explanation $\mu$
9:         else
10:            $\nu \leftarrow$ ExtractDualXp$(\mathcal{F} \setminus \mu, \kappa, \mathbf{v}, c)$
11:            $\mathbb{E}_{1-\rho} \leftarrow \mathbb{E}_{1-\rho} \cup \{\nu\}$           ▷ Collect dual explanation $\nu$
12:        if IsSwitchNeeded$(\mathbb{E}_\rho, \mathbb{E}_{1-\rho}, w, \alpha, \varepsilon)$ then
13:            $\rho \leftarrow 1 - \rho$                                  ▷ Flip phase of MinimalHS
       return $\mathbb{E}_1$, $\mathbb{E}_0$                               ▷ Result AXp's and CXp's
```

▶ Remark 11. Flipping enumeration phase ρ can be seamlessly done because we apply pure SAT-based hitting enumeration [12] where both $\mathbb{E}_\rho$ and $\mathbb{E}_{1-\rho}$ are represented as sets of *negative* and *positive* blocking clauses, respectively. As such, by instructing the SAT solver to opt for minimal or maximal models,[2] we can flip from computing hitting sets of $\mathbb{E}_{1-\rho}$ subject to $\mathbb{E}_\rho$ to computing hitting sets of $\mathbb{E}_\rho$ subject to $\mathbb{E}_{1-\rho}$, and vice versa. Importantly, this can be done while incrementally keeping the internal state of the SAT solver, i.e. no learnt information gets lost after the phase switch. Also, note that although the algorithm allows us to apply phase switching multiple times, our practical implementation switches *once* because AXp enumeration normally gets done much earlier than CXp enumeration, i.e. there is no point in switching back.

4 Experimental Results

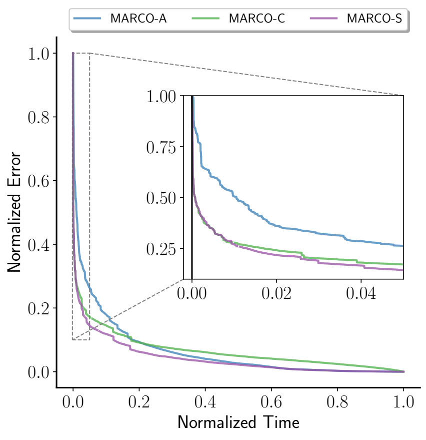
This section evaluates the proposed approach to anytime FFA approximation for the gradient boosted tree (BT) ML models on various publicly available data using a range of metrics. Here we report the results integrating all the experimental data averaged across all data instances in Section 4.2. The results for individual datasets can be found in Section 4.3.

4.1 Experimental Setup

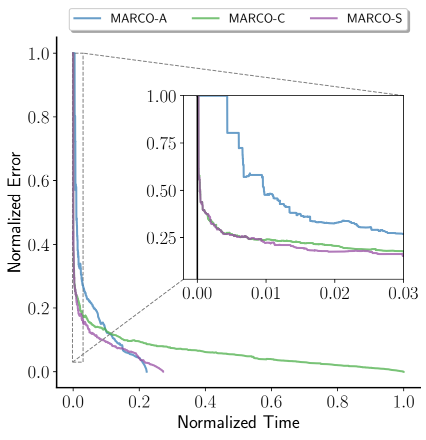
The experiments were performed on an Intel Xeon 8260 CPU running Ubuntu 20.04.2 LTS, with the 8GByte memory limit.

Prototype Implementation. The proposed approach was prototyped as a set of Python scripts, building on the approach of [56]. The proposed approach is referred to as MARCO-S, where the MARCO algorithm switches from CXp to AXp enumeration based on the conditions (6)–(7). The policy we use is to switch if either condition holds as we found

[2] Recall that in SAT solving, a *minimal* model is a satisfying assignment that respects subset-minimality wrt. the set of positive literals, i.e. where none of the 1's can be replaced by a 0 such that the result is still a satisfying assignment [5]. *Maximal* models can be defined similarly wrt. subset-minimality of negative literals.



(a) Mean.



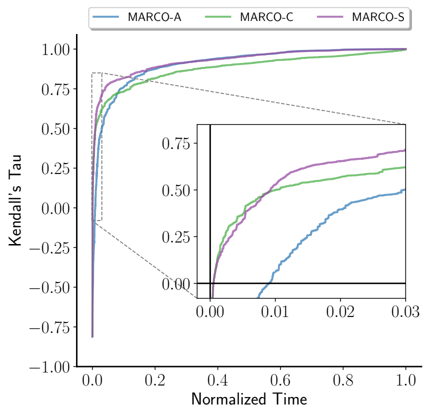
(b) Median.

Figure 3 FFA approximation error over time.

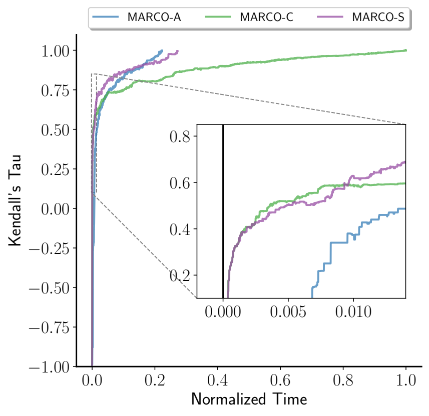
examples where each individual criterion was poor. For this, "sliding windows" of size $w = 50$ are used. Parameter α is set as $\alpha = 2$ in (6) to signify the extent by which the size of AXp's should be larger than the size of CXp's while parameter $\varepsilon = 1$ in (7) denoting the stability of the average CXp size.

Datasets and Machine Learning Models. The experiments include five well-known image and text datasets. We use the widely known *MNIST* [10, 44] dataset, which features handwritten digits from 0 to 9, with two concrete binary classification problems created: 1 vs. 3 and 1 vs. 7. Note that we treat MNIST "1vs3" and MNIST "1vs7" as two different datasets. Also, we consider the image dataset *PneumoniaMNIST* [55] differentiating normal X-ray cases from the cases of pneumonia. Since extracting *exact* FFA values for aforementioned image datasets turns out to be hard [56], we perform a size reduction, downscaling these images from $28 \times 28 \times 1$ to $10 \times 10 \times 1$. Additionally, 2 text datasets are considered in the experiments: *Sarcasm* [40, 41] and *Disaster* [14]. The *Sarcasm* dataset contains news headlines collected from websites, along with binary labels indicating whether each headline is sarcastic or non-sarcastic. The *Disaster* dataset consists of the contents of tweets with labels about whether a user announces a real disaster or not. The five considered datasets are randomly divided into 80% training and 20% test data. To evaluate the performance of the proposed approach, 15 test instances in each test set are randomly selected. Therefore, the total number of instances used in the experiments is 75. In the experiments, gradient boosted trees (BTs) are trained by XGBoost [8], where each BT consists of 25 to 40 trees of depth 3 to 5 per class. Test accuracy for *MNIST* (both "1vs3" and "1vs7"), *PneumoniaMNIST*, *Sarcasm*, and *Disaster* datasets is 0.99, 0.83, 0.69, and 0.74, respectively.

Competitors and Metrics. We compare the proposed approach (MARCO-S) against the original MARCO algorithms targeting AXp (MARCO-A) or CXp (MARCO-C) enumeration. We evaluate the FFA generated by these approaches by comparing it to the *exact* FFA through a variety of metrics, including errors, Kendall's Tau [22], rank-biased overlap (RBO) [54], and Kullback–Leibler (KL) divergence [24]. The *error* is quantified using Manhattan distance, i.e. the sum of absolute differences across all features in an instance. The comparison of feature ranking is assessed by Kendall's Tau and RBO, while feature distributions are compared

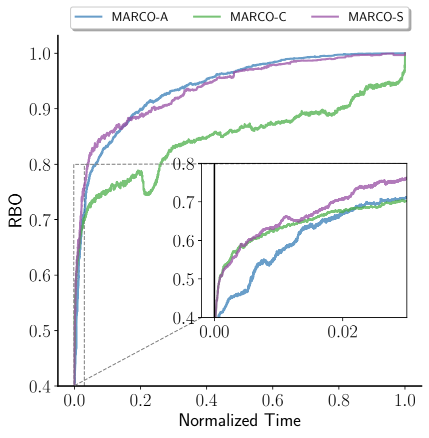


(a) Mean.

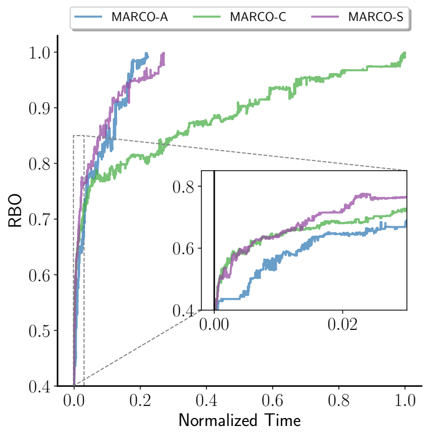


(b) Median.

Figure 4 Kendall's Tau over time.



(a) Mean.

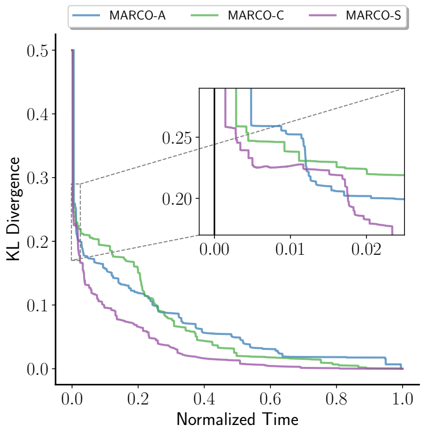


(b) Median.

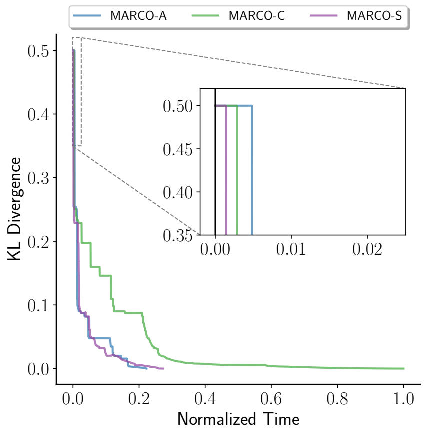
Figure 5 RBO over time.

by KL divergence.[3] Kendall's Tau and RBO produce values within the range of $[-1, 1]$ and $[0, 1]$, respectively, where higher values in both metrics indicate stronger agreement or similarity between the approximate FFA and the exact FFA. KL-divergence ranges from 0 to ∞, with the value approaching 0 reflecting better alignment between approximate FFA distribution and the exact FFA distribution. Note that if a feature in the exact FFA distribution holds a non-zero probability but is assigned a zero probability in the approximate one, the KL value becomes ∞. Finally, we also compare the efficiency of generating AXp's in the aforementioned approaches.

[3] Kendall's Tau is a correlation coefficient metric that evaluates the ordinal association between two ranked lists, providing a similarity measure for the order of values, while RBO quantifies similarity between two ranked lists, considering both the order and depth of the overlap. KL-divergence measures the statistical distance between two probability distributions.

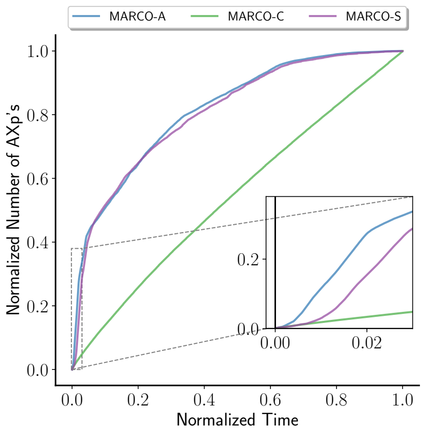


(a) Mean.

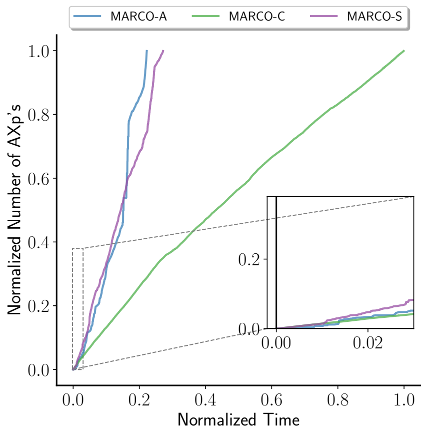


(b) Median.

Figure 6 KL divergence over time.



(a) Mean.



(b) Median.

Figure 7 Number of AXp's over time.

Table 1 The absolute numbers of MUSes (AXp's) and MCSes (CXp's) for different datasets.

(a) Numbers of MUSes/AXp's.

Dataset	Min	Mean	Max
MNIST1vs3	2916	15780.87	46576
MNIST1vs7	461	4028.27	10790
PneumoniaMNIST	21	8802.87	30996
Sarcasm	1056	12542.13	20024
Disaster	128	22853.20	35804

(b) Numbers of MCSes/CXp's.

Dataset	Min	Mean	Max
MNIST1vs3	992	17158.07	55108
MNIST1vs7	394	3558.80	9228
PneumoniaMNIST	30	6148.67	42308
Sarcasm	73	487.73	641
Disaster	88	4792.20	7028

4.2 Overview of Experimental Results

This section compares the proposed approach against the original MARCO algorithms for both AXp enumeration and CXp enumeration within the examined datasets. Figures 3 to 7 illustrate the results of approximate FFA in terms of the five aforementioned metrics, namely, errors, Kendall's Tau, RBO, KL divergence, and the number of AXp's. These results are obtained by averaging values across all instances. Note that since KL-divergence is ∞ when there exists a feature in the exact FFA distribution that holds a non-zero probability but is assigned a zero probability in the approximate one, to address this issue we assign 0.5 as the KL-divergence value instead of ∞ in this case.[4] The average runtime to extract exact FFA is 3255.30s (from 2.15s to 29191.42s), 19311.87s (from 9.39s to 55951.57s), and 3509.50s (from 9.26s to 30881.55s) for MARCO-A, MARCO-C, and MARCO-S, respectively. Switching from CXp to AXp enumeration in MARCO-S occurs on average at 106.77s. Note that since MARCO-A and MARCO-S tend to finish the enumeration process much earlier than MARCO-C, we also plot the median information because it better reflects the typical performance of these approaches in practice (which may be hard to see on the average data). Since the runtime required to get exact FFA varies, we normalized the runtime in each instance into [0, 1], where the longest time across three compared approaches in each instance is normalized to 1. Furthermore, we normalized the number of AXp's in each instance into the interval of [0, 1], as Table 1a shows that the numbers of AXp's (MUSes) vary across different instances and datasets. (Similarly, Table 1b indicates that the numbers of CXp's (MCSes) also differ across instances and datasets.) FFA approximation errors are also normalized into [0, 1] for each instance. Finally, switching from CXp to AXp enumeration in MARCO-S occurs at the time point of 0.0055 on the normalized scale (recall that it equals $\approx$106.77s).

Approximation Errors. Figure 3 displays the average and median errors of approximate FFA across all instances over time. Observe that in the early period, MARCO-C obtains more accurate approximate FFA regarding errors compared with MARCO-A, while beyond the 0.02 time fraction, the latter surpasses the former and eventually achieves 0 error faster, which also indicates that MARCO-A requires less time to acquire the exact FFA. Motivated by the above observation, the proposed approach aims at replicating the "best of two worlds" during the FFA approximation process. Observe that MARCO-S commences with CXp enumeration and so replicates the superior behavior of MARCO-C during the initial stage. Over time, MARCO-S triggers a switch criterion and transitions to targeting AXp's, thus emulating the behavior of the better competitor beyond the early stage, i.e. MARCO-A. Finally, MARCO-S acquires FFA with 0 error (i.e. exact FFA) as efficiently as MARCO-A.

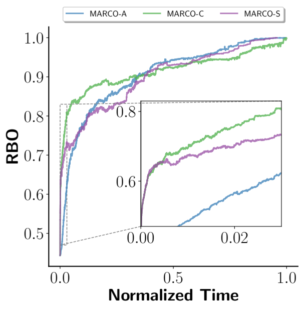
Feature Ranking. The results of Kendall's Tau and RBO are depicted in Figures 4 and 5. Initially, MARCO-C outperforms MARCO-A in terms of both feature ranking metrics. As time progresses, MARCO-A starts to surpass MARCO-C since 0.01 time fraction until the point of acquiring the exact FFA. Figures 4 and 5 demonstrate that initially MARCO-S manages to keep close to the better performing MARCO-C. When MARCO-A starts dominating, MARCO-S switches the target phase from CXp's to AXp's, replicating the superior performance displayed by MARCO-A.

Distribution. Figure 6 depicts the average and median results of KL divergence over time. Similar to feature ranking, MARCO-C is initially capable of computing an FFA distribution closer to the exact FFA distribution. Beyond the initial stage, MARCO-A exhibits the ability to generate closer FFA distribution. Once again, MARCO-S replicates the superior behavior between MARCO-A and MARCO-C most of the time. During the initial stage, MARCO-S reproduces the behavior of MARCO-C, and switch to target AXp's directly when the switch criterion is met. Surprisingly, MARCO-S *outperforms both competitors* throughout (almost) the entire time interval.

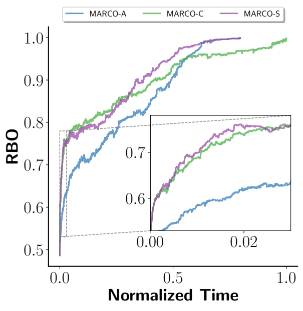
[4] According the experimental results we obtained, the maximum of *non-infinity* KL-divergence values is not greater than 0.5.

Table 2 Average runtime(s) in each dataset.

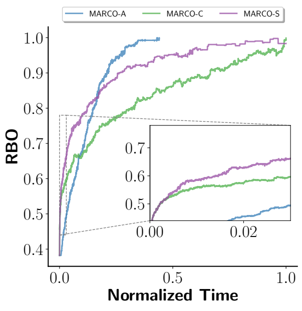
Approach	MNIST-1vs3	MNIST-1vs7	Pneumoniamnist	Sarcasm	Disaster
MARCO-A	9350.79	2844.15	1972.41	669.91	1439.24
MARCO-C	14787.22	7412.40	8343.55	33391.29	32624.89
MARCO-S	9970.55	2959.15	2016.49	975.31	1626.01



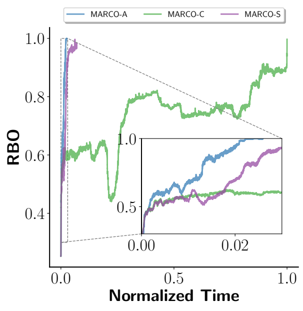
(a) MNIST-1vs3.



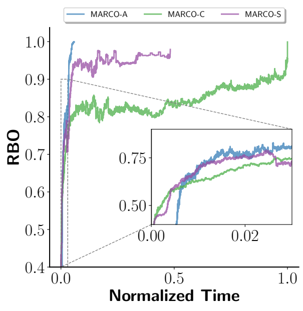
(b) MNIST-1vs7.



(c) PneumoniaMNIST.



(d) Sarcasm.

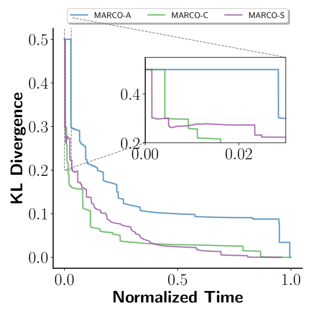


(e) Disaster.

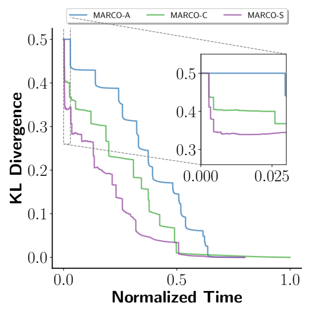
Figure 8 Mean RBO over time in each dataset.

Number of AXp's. The average and median results of the normalized number of AXp's are illustrated in Figure 7. MARCO-A generates AXp's faster and finishes earlier than MARCO-C. Observe that the proposed approach MARCO-S is able to avoid the inferior performance between MARCO-A and MARCO-C throughout the process. Initially, MARCO-S replicates the behavior of MARCO-C and then switches to target AXp's to replicate the performance of MARCO-A.

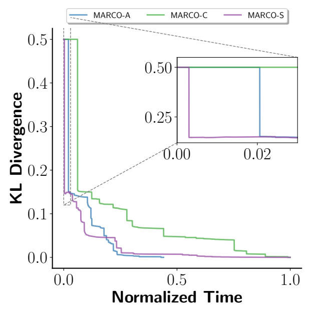
Summary. MARCO-S can replicate the behavior of the superior competitor for most of the computation duration, leading to efficient and good approximation of FFA. As illustrated by Figures 3–6 in terms of FFA errors, Kendall's Tau, RBO, and KL divergence, starting from CXp enumeration and switching to AXp enumeration based on the criteria (6)–(7) successfully replicates the behavior of the winning configuration of MARCO, thus getting close to the *virtual best solver*. Although in terms of the number of AXp's shown in Figure 7 MARCO-A consistently outperforms MARCO-C, those AXp's are not diverse enough to allow MARCO-A to beat MARCO-C in other relevant metrics. This is alleviated by MARCO-S, which manages to get enough diverse AXp's initially and then switches to target AXp's to catch up with the performance of MARCO-A.



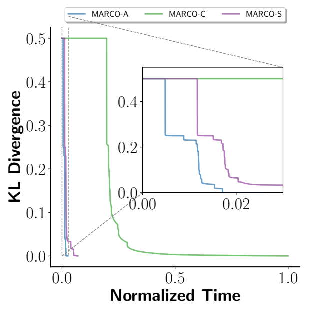
(a) MNIST-1vs3.



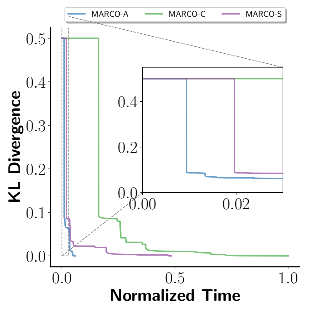
(b) MNIST-1vs7.



(c) PneumoniaMNIST.

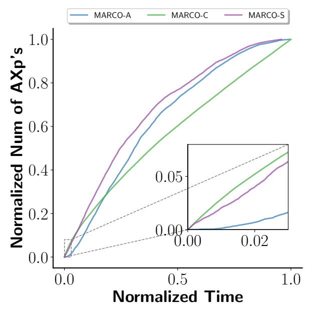


(d) Sarcasm.

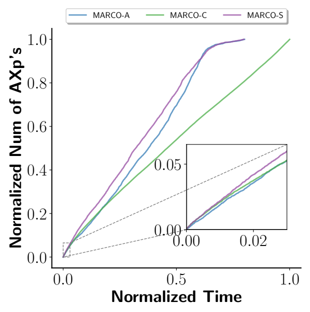


(e) Disaster.

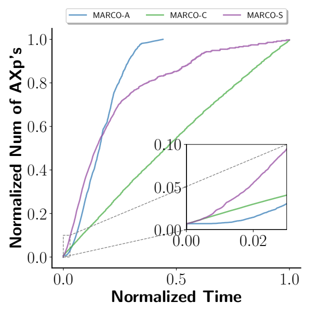
Figure 9 Mean KL-divergence over time in each dataset.



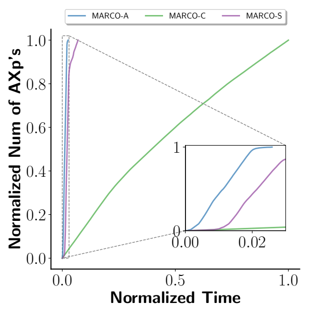
(a) MNIST-1vs3.



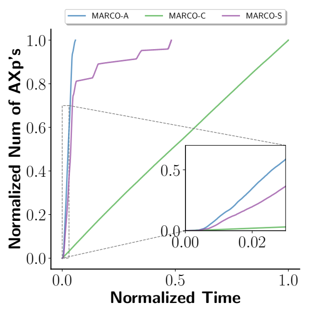
(b) MNIST-1vs7.



(c) PneumoniaMNIST.

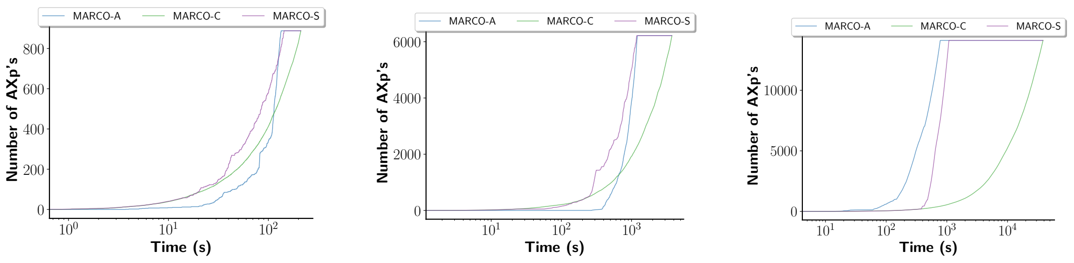


(d) Sarcasm.



(e) Disaster.

Figure 10 Mean number of AXp's over time in each dataset.



(a) MNIST-1vs7 instance. **(b)** PneumoniaMNIST instance. **(c)** Sarcasm instance.

Figure 11 Number of AXp's over time in example instances.

4.3 Detailed Experimental Results

This section compares the proposed approach (MARCO-S against the original MARCO algorithms for targeting AXp's (MARCO-A) and CXp's (MARCO-C) in each considered dataset. Figures 8 to 10 depict the average results of the comparison between the approximate FFA and the exact FFA using 3 metrics, namely, RBO, KL divergence, and the number of AXp's. The results show the mean values across 15 selected instances in a dataset. The average runtime of the three methods to acquire the exact FFA in each datadset is summarized in Table 2.

Feature Ranking. Figure 8 illustrates the results of RBO in each dataset. Observe that in all datasets but *Sarcasm*, MARCO-C performs better initially than MARCO-A, except in the *Sarcasm* dataset. Over time, MARCO-A gradually overtakes MARCO-C until reaching the point of obtaining the exact FFA. This figure demonstrates that MARCO-S maintains close to the superior performance exhibited by MARCO-C initially and then switches to targeting AXp's, replicating the superior performance demonstrated by MARCO-A. Nevertheless, in the *Sarcasm* dataset, MARCO-A consistently displays the superior performance. In the *Sarcasm* dataset, switching from CXp to AXp enumeration beyond the initial stage avoids reproducing the inferior performance between MARCO-A and MARCO-C in most of time.

Distribution. The average results of KL divergence over time are depicted in Figure 9. MARCO-C is initially capable of generating an FFA distribution more similar to the exact FFA distribution in *MNIST-1vs3* and *MNIST-1vs7* datasets. Afterwards, MARCO-A exhibits the ability to compute FFA distribution more similar to the exact FFA attribution. However, MARCO-A consistently generate a closer FFA distribution than MARCO-C in the other datasets. Once again, MARCO-S emulates the superior behavior between MARCO-A and MARCO-C in most of time or avoids replicating the inferior performance for a long time due to the switch mechanism. MARCO-S initially reproduces the behavior of MARCO-C, and switches to target AXp's when meeting the switch criterion. Surprisingly, MARCO-S exhibits the best performance among the competitors in most of the entire time interval in *MNIST-1vs3* and *MNIST-1vs7*.

Number of AXp's. Figure 10 shows the average results of the normalized number of AXp's in each dataset. Observe that compared with MARCO-A, MARCO-C is capable of generating AXp's more efficiently during the early stage in *MNIST-1vs3* and *MNIST-1vs7* datasets, but MARCO-A starts to outperform MARCO-C as time progresses. In the other three datasets, MARCO-A achieves similar or better performance in the entire process. As demonstrated by Figure 10, the proposed approach MARCO-S is able to avoid the inferior performance between MARCO-A and MARCO-C for most of the duration. Initially, MARCO-S emulates the behavior of MARCO-C, and transitions to target AXp's to replicate the performance of MARCO-A afterwards, preventing the reproduction of inferior performance. Remarkably, in the *MNIST-1vs7* dataset, MARCO-S emerges as the best-performing approach for most of time. Figure 11 presents numbers of AXp's over time in three example instances, demonstrating that MARCO-S can avoid the inferior performance between MARCO-A and MARCO-C for most of time in these three instances.

Summary. In alignment with the results presented in Section 4.2, MARCO-S is able to replicate the behavior of the superior competitor between MARCO-A and MARCO-C throughout most of the computation period, resulting in fast and good approximation of FFA. Figures 8 and 9 display that switching from CXp to AXp enumeration based on criteria 6–7 can reproduce the performance of the top MARCO configuration, closely approaching their virtual best solver. While MARCO-A consistently exhibits better than MARCO-C in some datasets in terms of the number of AXp's depicted in Figure 10, the lack of diversity among these AXp's prevents MARCO-A from outperforming MARCO-C in other relevant metrics. MARCO-S addresses this diversity issue by initially obtaining a diverse set of AXp's and then transitioning to targeting them, thereby matching the performance of MARCO-A.

5 Conclusions

Formal feature attribution (FFA) defines a crisp and easily understood notion of feature importance to a decision. It builds on the concepts of formal abductive and contrastive explanations [36], which can be related to the concepts of minimal unsatisfiable subsets (MUSes) and minimal correction subsets (MCSes) in the context of SAT solving. Unfortunately, for many classifiers and datasets FFA is challenging to compute exactly. As our paper demonstrates, it remains hard even if the set of CXp's is provided. Hence, there is a need for *anytime* approaches to compute FFA. One approach to compute and approximate FFA values is by exploiting the duality between AXp's and CXp's and applying the MARCO-style algorithms [45, 27, 29] of exhaustive AXp (resp., MUS) and CXp (resp., MCS) enumeration. As exhaustive explanation enumeration can be done by targeting either AXp's or CXp's, it is not always clear which approach is more efficient in practice from the perspective of the raw number of explanations but also from the view of the quality of FFA value approximations. Surprisingly, using CXp enumeration to generate AXp's leads to fast good approximations of FFA, but in the longer term it is worse than simply enumerating AXp's. This paper shows how to combine the approaches by diligently switching the phase of enumeration, without losing information computed in the underlying MARCO enumeration algorithm. This gives a highly practical approach to computing FFA.

The proposed mechanism can be readily adapted to a multitude of other problems, e.g. in the domains of over-constrained systems or model-based diagnosis (MBD), where one wants to collect a *diverse* and representative set of MUSes or explanations as the same minimal hitting set duality exists in unsatisfiability and MBD between the concepts of MUSes and MCSes, and explanations and diagnoses, respectively [6, 48].

References

1 Fahiem Bacchus and George Katsirelos. Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In *CAV*, pages 70–86, 2015.
2 James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186, 2005.

3 Anton Belov, Ines Lynce, and Joao Marques-Silva. Towards efficient MUS extraction. *AI Commun.*, 25(2):97–116, 2012.

4 Jaroslav Bendík, Ivana Cerná, and Nikola Benes. Recursive online enumeration of all minimal unsatisfiable subsets. In *ATVA*, pages 143–159, 2018.

5 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability: Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

6 Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1):25–46, 2003.

7 Karthekeyan Chandrasekaran, Richard M. Karp, Erick Moreno-Centeno, and Santosh S. Vempala. Algorithms for implicit hitting set problems. In *SODA*, pages 614–629, 2011.

8 Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, pages 785–794, 2016.

9 Peter Clark and Robin Boswell. Rule induction with CN2: some recent improvements. In *EWSL*, pages 151–163, 1991.

10 Li Deng. The MNIST database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

11 Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

12 Enrico Giunchiglia and Marco Maratea. Solving optimization problems with DLL. In *ECAI*, pages 377–381, 2006.

13 Éric Grégoire, Yacine Izza, and Jean-Marie Lagniez. Boosting MCSes enumeration. In *IJCAI*, pages 1309–1315, 2018.

14 Addison Howard, Devrishi, Phil Culliton, and Yufeng Guo. Natural language processing with disaster tweets, 2019. URL: `https://kaggle.com/competitions/nlp-getting-started`.

15 Xuanxiang Huang, Martin C. Cooper, António Morgado, Jordi Planes, and João Marques-Silva. Feature necessity & relevancy in ML classifier explanations. In *TACAS (1)*, pages 167–186, 2023.

16 Xuanxiang Huang and João Marques-Silva. The inadequacy of Shapley values for explainability. *CoRR*, abs/2302.08160, 2023.

17 Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NIPS*, pages 4107–4115, 2016.

18 Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976.

19 Alexey Ignatiev, Mikolas Janota, and Joao Marques-Silva. Quantified maximum satisfiability. *Constraints An Int. J.*, 21(2):277–302, 2016.

20 Alexey Ignatiev, Nina Narodytska, Nicholas Asher, and Joao Marques-Silva. From contrastive to abductive explanations and back again. In *AI*IA*, pages 335–355, 2020. `doi:10.1007/978-3-030-77091-4_21`.

21 Alexey Ignatiev, Nina Narodytska, and Joao Marques-Silva. Abduction-based explanations for machine learning models. In *AAAI*, pages 1511–1519, 2019. `doi:10.1609/AAAI.V33I01.33011511`.

22 Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

23 Ron Kohavi. Scaling up the accuracy of naive-Bayes classifiers: A decision-tree hybrid. In *KDD*, pages 202–207, 1996.

24 Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

25 Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. Interpretable decision sets: A joint framework for description and prediction. In *KDD*, pages 1675–1684. ACM, 2016.

26 Mark Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, pages 160–175, 2013.

27 Mark Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In *CPAIOR*, pages 160–175, 2013.

28 Mark H. Liffiton, Maher N. Mneimneh, Ines Lynce, Zaher S. Andraus, Joao Marques-Silva, and Karem A. Sakallah. A branch and bound algorithm for extracting smallest minimal unsatisfiable subformulas. *Constraints An Int. J.*, 14(4):415–442, 2009.

29 Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.

30 Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *SAT*, pages 173–186, 2005.

31 Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

32 Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *NeurIPS*, pages 4765–4774, 2017. URL: `https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html`.

33 Joao Marques-Silva, Thomas Gerspacher, Martin C. Cooper, Alexey Ignatiev, and Nina Narodytska. Explanations for monotonic classifiers. In *ICML*, pages 7469–7479, 2021. URL: `http://proceedings.mlr.press/v139/marques-silva21a.html`.

34 Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI*, pages 615–622, 2013.

35 João Marques-Silva and Xuanxiang Huang. Explainability is NOT a game. *CoRR*, abs/2307.07514, 2023.

36 João Marques-Silva and Alexey Ignatiev. Delivering trustworthy AI through formal XAI. In *AAAI*, pages 12342–12350, 2022.

37 Carlos Mencia, Alexey Ignatiev, Alessandro Previti, and Joao Marques-Silva. MCS extraction with sublinear oracle queries. In *SAT*, pages 342–360, 2016.

38 Carlos Mencia, Alessandro Previti, and Joao Marques-Silva. Literal-based MCS extraction. In *IJCAI*, pages 1973–1979, 2015.

39 Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artif. Intell.*, 267:1–38, 2019. `doi:10.1016/J.ARTINT.2018.07.007`.

40 Rishabh Misra and Prahal Arora. Sarcasm detection using news headlines dataset. *AI Open*, 4:13–18, 2023.

41 Rishabh Misra and Jigyasa Grover. *Sculpting Data for ML: The first act of Machine Learning.* Independently published, January 2021.

42 Vinod Nair and Geoffrey Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, pages 807–814, 2010.

43 Nina Narodytska, Nikolaj Bjørner, Maria-Cristina V. Marinescu, and Mooly Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361, 2018.

44 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

45 Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013.

46 Alessandro Previti, Carlos Mencía, Matti Järvisalo, and João Marques-Silva. Premise set caching for enumerating minimal correction subsets. In *AAAI*, pages 6633–6640, 2018.

47 J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.

48 Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. `doi:10.1016/0004-3702(87)90062-2`.

49 Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why should I trust you?": Explaining the predictions of any classifier. In *KDD*, pages 1135–1144, 2016. doi:10.1145/2939672.2939778.

50 Ronald L. Rivest. Learning decision lists. *Mach. Learn.*, 2(3):229–246, 1987.

51 Andy Shih, Arthur Choi, and Adnan Darwiche. A symbolic approach to explaining Bayesian network classifiers. In *IJCAI*, pages 5103–5111, 2018. doi:10.24963/IJCAI.2018/708.

52 Salil Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM J. Comput.*, 31(2):398–427, 2001.

53 L. G. Valiant. The complexity of computing the permanent. *Theoret. Comput. Sci.*, 8(2):189–201, 1979.

54 William Webber, Alistair Moffat, and Justin Zobel. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)*, 28(4):1–38, 2010.

55 Jiancheng Yang, Rui Shi, Donglai Wei, Zequan Liu, Lin Zhao, Bilian Ke, Hanspeter Pfister, and Bingbing Ni. MedMNIST v2-a large-scale lightweight benchmark for 2D and 3D biomedical image classification. *Scientific Data*, 10(1):41, 2023.

56 Jinqiang Yu, Alexey Ignatiev, and Peter J. Stuckey. On formal feature attribution and its approximation. *CoRR*, abs/2307.03380, 2023. arXiv:2307.03380, doi:10.48550/arXiv.2307.03380.

Small Unsatisfiable k-CNFs with Bounded Literal Occurrence

Tianwei Zhang
Algorithms and Complexity Group, TU Wien, Austria

Tomáš Peitl
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

We obtain the smallest unsatisfiable formulas in subclasses of k-CNF (exactly k distinct literals per clause) with bounded variable or literal occurrences. Smaller unsatisfiable formulas of this type translate into stronger inapproximability results for MaxSAT in the considered formula class. Our results cover subclasses of 3-CNF and 4-CNF; in all subclasses of 3-CNF we considered we were able to determine the smallest size of an unsatisfiable formula; in the case of 4-CNF with at most 5 occurrences per variable we decreased the size of the smallest known unsatisfiable formula. Our methods combine theoretical arguments and symmetry-breaking exhaustive search based on SAT Modulo Symmetries (SMS), a recent framework for isomorph-free SAT-based graph generation. To this end, and as a standalone result of independent interest, we show how to encode formulas as graphs efficiently for SMS.

2012 ACM Subject Classification Mathematics of computing → Solvers; Mathematics of computing → Graph enumeration; Theory of computation → Automated reasoning; Hardware → Theorem proving and SAT solving

Keywords and phrases k-CNF, (k, s)-SAT, minimally unsatisfiable formulas, symmetry breaking

Digital Object Identifier 10.4230/LIPIcs.SAT.2024.31

Related Version *Full Version*: `https://arxiv.org/abs/2405.16149` [28]

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.11282310`

Funding The project leading to this publication has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101034440 , and was supported by the Austrian Science Fund (FWF) within the project 10.55776/P36688.

1 Introduction

A (k, s)-formula is a propositional CNF formula in which each clause has exactly k distinct literals and each variable occurs (positively or negatively) in at most s clauses. Since Tovey [27] initiated the study of (k, s)-CNF formulas in 1984, they have been the subject of intensive investigation [3, 4, 7, 9, 10, 11, 12, 15, 21, 25]. Using Hall's Marriage Theorem, Tovey showed that all $(3, 3)$-CNF formulas are satisfiable, but allowing a fourth occurrence per variable yields a class of formulas for which the satisfiability problem is NP-complete. Kratochvíl, et al. [21] generalized this result and showed that for each $k \geq 3$, there exists a threshold $s = f(k)$ such that all $(k, f(k))$-formulas are satisfiable and checking the satisfiability of $(k, f(k) + 1)$-formulas (the *(k, s)-SAT problem*) is NP-complete. Therefore, determining whether (k, s)-SAT is NP-hard boils down to identifying an unsatisfiable (k, s)-formula. While tight asymptotic bounds for the threshold have been obtained [10], exact values are only known for $k \leq 4$ [11]: $f(3) = 3$, $f(4) = 4$; $f(5) \in [5, 7]$. No decision procedure is known for determining the threshold $f(k)$, since no upper bound on the size of a smallest unsatisfiable (k, s)-formula is known for $k > 4$.

It is an intriguing question of extremal combinatorics to determine the size of the smallest unsatisfiable (k, s)-formulas (with $s > f(k)$), and this paper sets out to address this question for various values of the parameters. This requires proving lower and upper bounds – an upper bound typically consists in exhibiting a formula with suitable parameters, while a lower bound requires a proof that no such formulas of a particular size exist. For $k = 3$, the unsatisfiable $(3, 4)$-formula constructed by Tovey [27] has 40 clauses, this was later improved to 19 and then 16 by Berman, et al. [3, 4].

A simple counting argument shows that such a formula must contain at least 8 clauses, hence there is a significant gap between the known lower and upper bounds. For $k = 4$, the gap is even larger. Stříbrná [26] constructed an unsatisfiable (4,5)-CNF formula with 449 clauses, which Knuth [20, p. 588] improved to 257. The same counting argument shows that such a formula must contain at least 16 clauses.

All constructions above have the following in common. A satisfiable (k, s)-formula with a backbone variable x that must be false in all satisfying truth assignments is first constructed. Such a formula is called a *(k, s)-enforcer.* Then one combines k copies of this formula together with a k-clause with the k backbone variables to obtain an unsatisfiable (k, s)-CNF formula. Aside from providing an upper bound for the size of a smallest unsatisfiable (k, s)-formula, the size of unsatisfiable (k, s)-enforcers has a direct effect on inapproximability results for certain NP-hard Max-SAT problems [3]. We also address the problem of the smallest size of a $(3, s)$ or $(3, p, q)$-enforcer in this paper.

Contribution

In this paper, we develop a general approach to computing small unsatisfiable (k, s)-formulas and (k, s)-enforcers. We also consider the more fine-grained setting of (k, p, q)-formulas, where p and q bound the number of positive and negative occurrences per variable, respectively. We observe a similar threshold phenomenon in the complexity of (k, p, q)-SAT as in the case of (k, s)-SAT (Lemma 4).

Our approach rests on utilizing the SAT Modulo Symmetries (SMS) framework [18, 16] for isomorph-free generation of unsatisfiable (k, s) and (k, p, q)-formulas for a fixed number n of variables and number m of clauses with parameters k, s, p, q.

The basic setting without any techniques for speed-up and divide-and-conquer scales up to about $m = 13$. We then use further theoretical arguments together with techniques for speed-up to determine the size of smallest formulas in various setting.

Smallest unsatisfiable formulas. We determined the size of smallest unsatisfiable $(3, s)$ and $(3, p, q)$-formulas for all possible s, p and q the values are given in Table 1 on the left for $(3, s)$-formulas and on the right for $(3, p, q)$-formulas.

In particular, we identified a smallest unsatisfiable $(3, 1, 3)$-formula with 22 clauses, which implies that $(3, 1, 3)$-SAT is NP-complete. Hence, we now have a direct and streamlined proof for the dichotomy of $(3, p, q)$-SAT (Theorem 5).

Smallest enforcers. Table 2 summarizes our results on the size of smallest $(3, s)$-enforcers and $(3, p, q)$-enforcers.

Table 1 Size of smallest unsatisfiable $(3, s)$-formulas (left) and size of smallest unsatisfiable $(3, p, q)$-formulas (right). ∞ indicates that for these parameters no unsatisfiable formula exists.

$s \leq 3$	$s = 4$	$s = 5$	$s \geq 6$
∞	16	11	8

	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q \geq 5$
$p = 1$	∞	∞	22	19	16
$p = 2$	–	20	11	10	10
$p \geq 3$	–	–	8	8	8

Table 2 Size of smallest $(3, s)$-enforcers (left) and smallest $(3, p, q)$-enforcers (right).

$s \leq 3$	$s = 4$	$s \geq 5$
∞	5	4

	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q \geq 5$
$p = 1$	∞	∞	7	6	5
$p = 2$	–	10	5	5	5
$p \geq 3$	–	–	5	5	5

A smaller unsatisfiable $(4, 5)$-formula. Recall that $f(4) = 5$. The smallest known unsatisfiable $(4, 5)$-formula is due to Knuth and has 257 clauses. We improve this by exhibiting an unsatisfiable $(4, 5)$-formula with 235 clauses. We obtain this by first computing an auxiliary formula with SMS and then constructing from it an unsatisfiable $(4, 5)$-formula by disjunctive splitting.

Structure of the paper. After this introduction, and preliminaries, the paper is organized into two main parts. In Section 3, we lay out the principle encoding that provides the basis for all our results. In Section 4, we dive into the more complicated, technical aspects that are necessary to rule out the existence of larger formulas and obtain better lower bounds. Lemmas and Theorems marked with $\star$ have proofs in the full version [28], where some arguments have been streamlined.

2 Preliminaries

For positive integers $k < \ell$, we write $[k] = \{1, 2, \dots, k\}$, $[-k] = \{-k, -(k-1), \dots, -1\}$, and $[k, \ell] = \{k, \dots, \ell\}$. We assume familiarity with fundamental notions of propositional logic [19]. In this paper we will talk about *(minimally) unsatisfiable* propositional formulas represented as graphs, specified by properties expressed as *quantified Boolean formulas*, and about *isomorphisms (symmetries)* of these formulas. We review the relevant basics below.

CNF formulas. A *literal* is a (propositional) variable x or a negated variable $\overline{x}$, whereby $\overline{\overline{x}} = x$. We write $\mathsf{var}(x) := \mathsf{var}(\overline{x}) := x$ for the variable belonging to a literal. A set S of literals is *tautological* if $S \cap \overline{S} \neq \emptyset$, where $\overline{S} = \{\, \overline{x} : x \in S \,\}$. A *clause* is a finite non-tautological set of literals. A k-clause is a clause that contains exactly k literals. The 0-clause is denoted by $\square$. A *(CNF) formula* is a finite set of clauses. For $k \geq 1$, a *k-CNF formula* is a formula in which all clauses are k-clauses (please note that some authors allow a k-CNF formula to contain clauses with fewer than k literals, but it is significant in our context that the number is exactly k) and a *$(\leq k)$-CNF formula* is a formula in which all clauses contain at most k literals. A variable x *occurs positively* in a clause C if $x \in C$, it *occurs negatively* in C if $\overline{x} \in C$, and it *occurs* in C if it occurs in C positively or negatively. For a literal x, $F[x]$ denotes the set of clauses in F in which $\mathsf{var}(x)$ occurs. We will often write a $\leq k$-CNF with m

clauses as a $k \times m$ matrix whose columns are the clauses, and entries are literal occurrences. When a clause has $r < k$ literals, we write $\times$ in the last $k - r$ rows in the corresponding column of the matrix.

Counting occurrences. For a clause C, we write $\mathsf{var}(C)$ for the set of variables that occur in C, and for a CNF formula F we write $\mathsf{var}(F) = \bigcup_{C \in F} \mathsf{var}(C)$. The *degree* of a variable in a formula F is defined as $\deg_F^{\mathsf{var}}(x) := |F[x]|$. For literals, we put $\deg_F^{\mathsf{var}}(\overline{x}) := \deg_F^{\mathsf{var}}(x)$. The *degree* of a literal x, denoted by $\deg_F(x)$, is the number of clauses in which the literal occurs. We may omit the subscript F when it is clear from the context.

For $k, s \geq 1$, a (k,s)*-formula* is a k-formula in which each variable occurs in at most s clauses, and a $(\leq k, s)$*-formula* is a $(\leq k)$-formula in which each variable occurs in at most s clauses. For $k, p, q \geq 1$, a (k,p,q)*-formula* is a k-formula in which each variable occurs in at most p clauses positively and in at most q clauses negatively, and a $(\leq k, p, q)$*-formula* is a $(\leq k)$-formula with the same constraint. Without loss of generality, we will assume $p \leq q$ for (k,p,q)-formulas as we can always swap positive and negative literals.

We define $\mu(k,s)$ to be the number of clauses of a smallest unsatisfiable (k,s)-formula, and $\mu(k,p,q)$ denotes that of the smallest unsatisfiable (k,p,q)-formula.

Bounded literal occurrence SAT. A *truth assignment* for a set X of variables is a mapping $\tau : X \to \{0,1\}$. In order to define τ on literals, we set $\tau(\overline{x}) = 1 - \tau(x)$. A truth assignment τ *satisfies* a clause C if C contains at least one literal x with $\tau(x) = 1$, and τ *satisfies* a formula F if it satisfies every clause of F. In the latter case, we call F *satisfiable.* The SATISFIABILITY problem (SAT) is to decide whether a given formula is satisfiable. (k,s)-SAT is SAT restricted to (k,s)-formulas, and (k,p,q)-SAT is SAT restricted to (k,p,q)-formulas.

Enforcers. A (k,s)-*enforcer* is a satisfiable (k,s)-formula F with a variable x with $\deg_F^{\mathsf{var}}(x) < s$ that is set to the same value in every satisfying assignment. A (k,p,q)-*enforcer* is a satisfiable (k,p,q) formula F with a variable x which is either set to true in every satisfying assignment, and then $\deg_F(x) < p$, or it is set to false in every satisfying assignment, and then $\deg_F(\overline{x}) < q$. We say the literal of x that is set to true in every satisfying assignment is *enforced.* An enforcer can be completed into an unsatisfiable $(\leq k, s)$ or $(\leq k, p, q)$-formula by adding the unit clause containing the negation of the enforced literal.

Minimal unsatisfiability. A CNF formula is *minimally unsatisfiable* if it is unsatisfiable but dropping any of its clauses results in a satisfiable formula. Let MU denote the class of all minimally unsatisfiable formulas. The *deficiency* of a CNF formula F is $\delta(F) = |C| - |\mathsf{var}(F)|$. It is known that $\delta(F) > 0$ for any $F \in \text{MU}$ [1]; therefore it is natural to parameterize MU by deficiency and to consider the classes $\text{MU}(d) := \{\, F \in \text{MU} : \delta(F) = d \,\}$ for $d \geq 1$.

Variable elimination. It is well-known that one can eliminate variables of a CNF formula by a process often called *DP-resolution*, after an algorithm of Davis and Putnam [6], as follows. For two clauses C, D with $x \in C, \overline{x} \in D$, the *resolution* rule yields the *resolvent* clause $C \cup D \setminus \{x, \overline{x}\}$. Let F be a CNF and $x \in \mathsf{var}(F)$. We define $F^x := F \setminus F[x] \cup \{C \cup D \setminus \{x, \bar{x}\} \mid C, D \in F; C \cap \overline{D} = \{x\}\}$. In other words, the result of eliminating x from F is the formula that contains all clauses where x does not occur together with all possible non-tautological resolvents on x. It is easy to see that $\exists x\, F$ and F^x are logically equivalent, and in particular, if F is unsatisfiable, so is F^x.

Blocked clauses. A clause C in a CNF F is *blocked in F on the literal $x \in C$* if for every $C' \in F$ with $\overline{x} \in C'$, there exists a variable $y \neq \mathsf{var}(x)$ with $y \in C, \overline{y} \in C'$ or $y \in C', \overline{y} \in C$. A clause is *blocked in F* if it is blocked on at least one of its literals. Blocked clauses are a fundamental SAT preprocessing technique: when C is blocked in F and F is satisfiable, then $F \cup \{C\}$ is also satisfiable [13, 22]; in other words, blocked clauses may be added or removed without impacting satisfiability (notice that this also follows from soundness of variable elimination). We will use the simple corollary that a minimally unsatisfiable formula cannot contain a blocked clause.

QBF. Quantified Boolean formulas generalize propositional logic with quantification. In this paper, we will need only the fragment of *closed prenex* 2*-QBFs* with one quantifier alternation. A 2-QBF has the form $\exists X \forall Y \, \Phi(X, Y)$, where X and Y are sets of propositional variables, and Φ is a propositional formula. A 2-QBF is *true* if there exists an assignment $\tau : X \to \{0, 1\}$ such that $\Phi(\tau(X), Y)$ evaluates to true for every assignment to Y, where $\tau(X)$ denotes the substitution of τ values for X into Φ. The formula is *false* if no such assignment τ exists.

Graphs. We only use undirected and simple graphs (i.e., without parallel edges or self-loops). A *graph* G consists of set $V(G)$ of vertices and a set $E(G)$ of edges; we denote the edge between vertices $u, v \in V(G)$ by uv or equivalently vu.

We write $\mathcal{G}_n$ to denote the class of all graphs with $V(G) = [n]$. The *adjacency matrix* A of a graph $G \in \mathcal{G}_n$ is the $n \times n$ $\{0, 1\}$-matrix where the element at row v and column u, denoted by $A(v, u)$, is 1 iff $vu \in E(G)$.

Isomorphisms. For a permutation $\pi : [n] \to [n]$, $\pi(G)$ denotes the graph obtained from $G \in \mathcal{G}_n$ by the permutation π, where $V(\pi(G)) = V(G) = [n]$ and $E(\pi(G)) = \{\, \pi(u)\pi(v) : uv \in E(G) \,\}$. Two graphs $G_1, G_2 \in \mathcal{G}_n$ are *isomorphic* if there is a permutation $\pi : [n] \to [n]$ such that $\pi(G_1) = G_2$; in this case G_2 is an *isomorphic copy* of G_1. A *partially defined graph* [17] is a graph G where $E(G)$ is split into two disjoint sets $D(G)$ and $U(G)$. $D(G)$ contains the *defined* edges, $U(G)$ contains the *undefined* edges. A (*fully defined*) graph is a partially defined graph G with $U(G) = \emptyset$. A partially defined graph G can be *extended* to a graph H if $D(G) \subseteq E(H) \subseteq D(G) \cup U(G)$.

CNF Formulas as graphs. For sets S, S', T, we write $T = S \uplus S'$ if $T = S \cup S'$ and $S \cap S' = \emptyset$. A *2-graph* is an undirected graph $G = (V, E)$ together with a partition of its vertex set into two disjoint *blocks* $V_1 \uplus V_2 = V$. Two 2-graphs $G = (V_1 \uplus V_2, E)$ and $G' = (V_1' \uplus V_2', E')$ are *isomorphic* if there exists a bijection $\phi : V_1 \uplus V_2 \to V_1' \uplus V_2'$ such that $v \in V_i$ if and only if $\phi(v) \in V_i'$, $i = 1, 2$, and $\{u, v\} \in E$ if and only if $\{\phi(u), \phi(v)\} \in E'$. The *clause-literal graph* of a CNF formula F is the 2-graph $G(F) = (V_1 \uplus V_2, E)$ with $V_1 = \mathsf{lit}(F)$, $V_2 = F$, and $E = \{\, \{x, \overline{x}\} : x \in \mathsf{var}(F) \,\} \cup \{\, \{C, \ell\} : C \in F, \ell \in C \,\}$. We refer to the edges $\{\, x, \overline{x} \,\}$ as *variable* edges. It is easy to verify that any two CNF formulas are isomorphic if and only if their clause-literal graphs are isomorphic (as 2-graphs). Note that we can safely assume that the first $|V_1|$ rows of the adjacency matrix of a clause-literal graph correspond to the vertices in V_1, and we will thoroughly do so throughout this paper.

SAT Modulo Symmetries (SMS). SMS [18] is a framework that augments a CDCL (conflict-driven clause learning) SAT solver [8, 23] with a custom propagator that can reason about symmetries, allowing to search modulo isomorphism for graphs in $\mathcal{G}_n$ satisfying a property specified in (quantified) propositional logic.

During search, the SMS propagator can trigger additional conflicts on top of ordinary CDCL and consequently learn *symmetry-breaking clauses*, which exclude isomorphic copies of graphs. More precisely, only those copies are kept which are lexicographically minimal (*canonical*) when considering the rows of the adjacency matrix concatenated into a single vector. A key component is a minimality check, which decides whether a partially defined graph can be extended to a minimal graph; if it cannot, a corresponding clause is learned. For a full description of SMS, we refer to the original work where the framework was introduced [18]. In SMS, it is possible to specify a partition of the vertex set and restrict the symmetry breaking to those permutations that preserve the partition. In Section 3.1, we explain how we can specify partitions to efficiently generate formulas (represented by 2-graphs) modulo isomorphism with SMS.

3 Basic encoding

In this section we explain the methodology all of our investigations build upon, and review results already obtainable with it. In the next section, we will delve into technical details and improvements that are necessary to scale up this basic approach.

The idea is to first list all possible number of clauses a smallest (k, s) or (k, p, q)-formula can have, and then decide whether an unsatisfiable formula exists with increasing m. For each m, we split the decision problem further by specifying number of variables n the sought formula has. For fixed m and n, we reduce this task to deciding the satisfiability of a suitable quantified Boolean formula and give it to QBF-enabled SMS. Because we gradually increase m, the first time we hit a satisfiable instance we know the formula thus produced is smallest possible. The desired QBF should have its models correspond to unsatisfiable (k, s)-formulas with n variables and m clauses. Since unsatisfiability is coNP-complete, we cannot hope to obtain a polynomial-size propositional encoding (unless NP = coNP), and instead we use a 2-QBF of the form $\exists X \forall Y \, \Phi^{n,m}_{k,s}(X) \wedge \neg\Sigma^{n,m}(X, Y)$ (or $\exists X \forall Y \, \Phi^{n,m}_{k,p,q}(X) \wedge \neg\Sigma^{n,m}(X, Y)$), where $\Phi^{n,m}_{k,s}(X)$ ($\Phi^{n,m}_{k,p,q}(X)$) expresses that X represents a (k, s)-formula ((k, p, q)-formula) with n variables and m clauses, and $\Sigma^{n,m}(X, Y)$ expresses that the assignment represented by Y satisfies X.

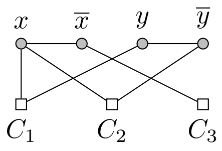
3.1 Hard-coding the first part of the clause-literal graph

Before we delve into the details of the encoding, we observe the following fact about the lexicographically minimal matrix of a clause-literal graph.

Let $\mathrm{pos}(i) = r$ if the rth row/column of the adjacency matrix represents the positive literal of the ith variable, and similarly for $\mathrm{neg}(i)$. The first block of a clause-literal graph contains the vertices corresponding to literals, i.e., we have $\mathrm{pos}(i), \mathrm{neg}(i) \in [2n]$ for all $i \in [n]$.

▶ **Theorem 1.** *The lexicographically minimal matrix of any clause-literal graph is antidiagonal in the upper-left (variables) block, i.e., for all $i, j \leq 2n$, $A(i, j) = 1$ iff $i + j = 2n + 1$. For the ordering of literals, this means that for each i we have $\{\mathrm{pos}(i), \mathrm{neg}(i)\} = \{j, 2n + 1 - j\}$ for some $j \in [n]$.*

Proof. Towards a contradiction, consider a lexicographically minimal adjacency matrix of some clause-literal graph, and the row i, where antidiagonality is first violated, because the 1-entry is in column $2n + 1 - j$ for $j > i$ (and not $j = i$ as it should be; notice that $j < i$ is impossible since the literal $2n + 1 - j$ would have to be adjacent to both j and i). Then, swapping the vertices $2n + 1 - j$ and $2n + 1 - i$ yields a lexicographically smaller matrix. ◀



	$\overline{x}$	$\overline{y}$	y	x	C_1	C_2	C_3
$\overline{x}$	0	0	0	1	0	0	1
$\overline{y}$		0	1	0	0	1	0
y			0	0	1	0	0
x				0	1	1	0
C_1					0	0	0
C_2						0	0
C_3							0

Figure 1 Consider the formula $F = \{C_1, C_2, C_3\}$ with $C_1 = \{x, y\}$, $C_2 = \{x, \overline{y}\}$, and $C_3 = \{\overline{x}\}$. We see the corresponding 2-graph and the upper part of its lexicographically minimal adjacency matrix. Observe how the left part is indeed antidiagonal.

Theorem 1 shows that we can hard-code the top-left $2n \times 2n$ matrix of the adjacency matrix. Doing so has the following benefits. The immediate advantage is that with fewer undecided variables to solve, SMS terminates more quickly. Also, this breaks the symmetries from reordering the literal nodes of the clause-literal matrix. Finally, as we will see more clearly in the next part, fixing the matching between literal nodes reduces the size of the encoding since we no longer need to describe the cardinality constraints on how many times a variable can occur *conditionally* on an undecided matching among literal nodes. Figure 1 shows an example of a clause-literal graph and its corresponding lexicographically minimal adjacency matrix.

3.2 Encodings for formulas

In this part, we describe the detailed construction of the specific 2-QBF we use, whose models correspond to unsatisfiable (k, s)-formulas with n variables and m clauses. We write our encoding in the standard circuit-QBF QCIR format [14].

We express cardinality constraints using cardinality networks [2]. A cardinality network Card_b^a takes b binary inputs and outputs the most significant a inputs ordered from more significant to less. For the purpose of illustration, suppose we want variable z to be true if and only if exactly d variables out of $x_1, \ldots, x_b$ are true. Let $(y_1, \ldots, y_d, y_{d+1}) := \mathrm{Card}_b^{d+1}(x_1, \ldots, x_b)$. Note that this way y_i is true if and only if there are at least i true inputs among $x_1, \ldots, x_b$. So we can define $z := y_d \wedge \neg y_{d+1}$. When the number of inputs and outputs is clear from the context, we just write Card. Let $i, i' \in [n], j < j' \in [m]$, we define $a_{i,i'} := A(\mathrm{pos}(i), \mathrm{pos}(i'))$, $a_{i,-i'} := A(\mathrm{pos}(i), \mathrm{neg}(i'))$ and $a_{-i,-i'} := A(\mathrm{neg}(i), \mathrm{neg}(i'))$ for edges between literal vertices, $e_{i,j} := A(\mathrm{pos}(i), j + 2n)$ and $e_{i,j} := A(\mathrm{neg}(i), j + 2n)$ for edges from a literal node to a clause node, and $c_{j,j'} := A(j + 2n, j' + 2n)$ for edges between clause nodes. Take a sufficiently large x. For all $i \in [n], t \in [n] \cup [-n]$ and $j \in [m]$, define

$$\begin{aligned}
(\omega_{t,1}, \omega_{t,2}, \ldots, \omega_{t,x}) &:= \mathrm{Card}(e_{t,1}, e_{t,2}, \ldots, e_{t,m}),\\
(\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,x}) &:= \mathrm{Card}(e_{i,1}, e_{i,2}, \ldots, e_{i,m}, e_{-i,1}, e_{-i,2}, \ldots, e_{-i,m}), \text{ and}\\
(\tau_{j,1}, \tau_{j,2}, \ldots, \tau_{j,k+1}) &:= \mathrm{Card}(e_{1,j}, e_{2,j}, \ldots, e_{n,j}, e_{-1,j}, e_{-2,j}, \ldots, e_{-n,j}).
\end{aligned}$$

The following are some useful properties expressed in propositional logic which we use as components in the desired 2-QBF. F_1 expresses that there are no edges between two clause nodes. F_2 expresses that each literal occurs at least once. F_3 expresses that a literal and its negation cannot occur in the same clause. F_4^v expresses that each variable occurs at most s times. F_5 expresses that each clause contains exactly k literals.

$$F_1 := \bigwedge_{0\le j<j'<m} \neg c_{j,j'}, \quad F_2 := \bigwedge_{i\in[n]\cup[-n]} \bigvee_{j\in[m]} e_{i,j}, \quad F_3 := \bigwedge_{i\in[n],j\in[m]} \neg e_{i,j} \vee \neg e_{-i,j}.$$

$$F_4^v := \bigwedge_{i\in[n]} \neg\sigma_{i,s+1}, \quad F_4^l := \bigwedge_{i\in[n]\cup[-n]} \neg\omega_{i,q+1} \wedge \bigwedge_{i\in[n]} \neg\omega_{i,p+1} \vee \neg\omega_{-i,p+1},$$

$$F_5 := \bigwedge_{j\in[m]} \tau_{j,k} \wedge \neg\tau_{j,k+1}.$$

It is easy to see that a smallest (k,s) or (k,p,q)-formula must be minimally unsatisfiable, and thus we can also require (in $\Phi^{n,m}_{k,s}$) that it contain no blocked clauses. Given $i \in [n] \cup [-n]$ and $j, j' \in [n]$, define $\psi_{i,j,j'}$ and $\varphi_{i,j}$ as follows. Here $\varphi_{i,j}$ expresses that the j-th clause is not blocked on the literal i.

$$\psi_{i,j,j'} := \bigwedge_{\substack{i'\in[n]\cup[-n]\\ i'\ne i,-i}} (\neg e_{i',j} \vee \neg e_{-i',j'}), \quad \varphi_{i,j} := \neg e_{i,j} \vee \bigvee_{\substack{j'\in[m]\\ j'\ne j}} (e_{-i,j'} \wedge \psi_{i,j,j'}).$$

F_6 expresses that the formula contained no blocked clauses.

$$F_6 := \bigwedge_{\substack{i\in[n]\cup[-n],\\ j\in[m]}} \varphi_{i,j}$$

F_7 hard-codes the matching between the literal nodes.

$$F_7 := \bigwedge_{i\in[n]} a_{i,-i} \wedge \bigwedge_{\substack{i\in[n],i'\in[-n]\\ i\ne -i'}} \neg a_{i,i'} \wedge \bigwedge_{i<i'\in[n]} \neg a_{i,i'} \wedge \bigwedge_{i'<i\in[-n]} \neg a_{i,i'}$$

Let $X := \{\, A(i,j) : i < j \in [2n+m] \,\}$ and $Y = \{\, \alpha_i : i \in [m] \,\}$. For $i \in [n]$, we put $\alpha_{-i} := \neg\alpha_i$. Finally, define $\Phi^{n,m}_{k,s}(X) := F_1 \wedge F_2 \wedge F_3 \wedge F_4^v \wedge F_5 \wedge F_6 \wedge F_7$, $\Phi^{n,m}_{k,p,q}(X) := F_1 \wedge F_2 \wedge F_3 \wedge F_4^l \wedge F_5 \wedge F_6 \wedge F_7$ and

$$\Sigma^{n,m}(X,Y) := \bigwedge_{j\in[m]} \bigvee_{i\in[n]\cup[-n]} \alpha_i \wedge e_{i,j}.$$

3.3 Preliminary findings

To determine the exact value of $\mu(3,s)$ and $\mu(3,p,q)$ for various choices of s, p and q, we enumerate all permissible values of (m,n), where m is the number of clauses and n is the number of variables, from smaller to bigger in the lexicographical order. For each pair (m,n), we solve the formula described in Section 3.2 with SMS.[1] We terminate the solver if it cannot answer within 5 days.

We ran the solver on a Sun Grid Engine (SGE) cluster consisting of heterogeneous machines running Ubuntu 18.04.6 LTS.[2]

We begin by observing that since allowing more occurrences yields a larger class of formulas, by definition, $\mu(k,s) \le \mu(k,s-1)$, $\mu(k,p,q) \le \min\big(\mu(k,p-1,q), \mu(k,p,q-1)\big)$, and $\mu(k,p+q) \le \mu(k,p,q)$. The following lemmas provide preliminary bounds on m and n for our enumeration.

[1] `https://sat-modulo-symmetries.readthedocs.io`
`https://github.com/markirch/sat-modulo-symmetries`

[2] The cluster contains nodes with the following architectures: 2× Intel Xeon E5540 with 2.53 GHz Quad Core, 2× Intel Xeon E5649 with 2.53 GHz 6-core, 2× Intel Xeon E5-2630 v2 with 2.60GHz 6-core, 2× Intel Xeon E5-2640 v4 with 2.40GHz 10-core and 2× AMD EPYC 7402 with 2.80GHz 24-core.

Table 3 Preliminary results based only on the method of this section, for $\mu(3,s)$ (left) and $\mu(3,p,q)$ (right). Since the right table is symmetric with respect to the main diagonal, we only give the upper triangle due to the assumption $q \geq p$. All values for $s \geq 6$ and $q \geq p \geq 3$ are 8: less is not possible by Lemma 2. Lemma 2 also rules out a $(3,2,q)$-formula with 9 clauses and $q \geq 5$.

$s=4$	$s=5$	$s \geq 6$
$[14,16]$	11	8

	$q=1$	$q=2$	$q=3$	$q=4$	$q \geq 5$
$p=1$	∞	∞	$[14,\infty]$	$[11,\infty]$	$[8,\infty]$
$p=2$	–	$[8,20]$	11	10	10
$p \geq 3$	–	–	8	8	8

▶ **Lemma 2.** *An unsatisfiable k-CNF formula that contains a variable of type (p,q) has at least $2^k + |q-p|$ clauses.*

Proof. Let F be an unsatisfiable k-CNF with m clauses, n variables, and x a variable of type (p,q). $F|_{\overline{x}} := \{\, C \setminus \{x\} \in F : \overline{x} \notin C \,\}$, obtained from F by setting x to false, is unsatisfiable, has $n-1$ variables, p clauses of size $k-1$, and $m-p-q$ clauses of size k. Since a clause of size r is falsified by 2^{n-1-r} assignments, and each assignment falsifies some clause, we have $p2^{n-k} + (m-p-q)2^{n-1-k} \geq 2^{n-1}$, and solving for m completes the proof. ◀

▶ **Lemma 3.** *A minimally unsatisfiable (k,s)-formula with m clauses has between $\lceil \frac{k \cdot m}{s} \rceil$ and $m-1$ variables. Similarly, a minimally unsatisfiable (k,p,q)-formula with m clauses has between $\lceil \frac{k \cdot m}{p+q} \rceil$ and $m-1$ variables.*

Proof. With n variables of degree $\leq s$ there are $mk \leq ns$ literal occurrences. For the upper bound, recall that minimally unsatisfiable formulas have positive deficiency. ◀

It is known that an unsatisfiable (3,4)-formula with 16 clauses and an unsatisfiable (3,2,2)-formula with 20 clauses exist [4]. We give the formulas as $E_{3,4}$ and $M_{3,2,2}$ in the appendix. The experimental results combined with this knowledge yield Table 3. One can find a smallest $(3,2,3)$, $(3,2,4)$ and $(3,3,3)$-formula in the appendix as $M_{3,2,3}$, $M_{3,2,4}$, and $M_{3,3,3}$. $M_{3,2,3}$ also serves as a smallest $(3,5)$-formula, and $M_{3,3,3}$ as a smallest $(3,6)$ formula.

A closer look at the time spent on deciding the existence of an unsatisfiable $(3,4)$-formula with different n and m shown in Table 4 reveals that this basic method reaches its limit with formulas of about 13 clauses. Thus, further considerations are called for if we want to determine the precise value for some of the entries in the tables.

Table 4 Time spent deciding the existence of an unsatisfiable $(3,4)$-formula with n variables and m clauses (without/with blocked-clause detection encoded as F_6). Unsolved queries are marked by to, blank areas are out of bounds determined by Lemma 3. All terminated queries were unsatisfiable except the one marked in blue with $n=12$ and $m=16$.

n	$m=8$	$m=9$	$m=10$	$m=11$	$m=12$
6	0.4s/0.6s				
7	0.7s/1.2s	1.9s/2.0s			
8		4.9s/5.9s	7.3s/10.4s		
9			42s/1m	1m/3m	5m/4m
10				12m/7m	43m/17m
11					3h/1.6h

n	$m=13$	$m=14$	$m=15$	$m=16$
10	2h/1h			
11	12h/6h	28h/27h		
12	5d/42h	to	to	to/5d
13		to	to	to
14			to	to
15				to

3.4 A dichotomy theorem for $(3, 1, q)$-SAT

The following extends a result by Kratochvíl, et al. [21, Lemma 2.2].

▶ **Lemma 4** ($\star$). *Let $k \geq 3$ and $p, q \geq 1$ such that $p + q \geq 3$. If there exists an unsatisfiable (k, p, q)-formula, then (k, p, q)-*SAT *is* NP*-hard.*

▶ **Theorem 5** (Dichotomy). *For any $p, q \geq 1$, if $p + q < 4$ then $(3, p, q)$-*SAT *is solvable in polynomial time, otherwise $(3, p, q)$-*SAT *is* NP*-hard.*

Proof. $(3, p, q)$-SAT is a special case of $(3, p + q)$-SAT, and $(3, s)$-SAT is in P for $s \leq 3$ [27]. Let $p + q \geq 4$, w.l.o.g., $p \leq q$. If $p = 1$, then $q \geq 3$, if $p \geq 2$, then $q \geq 2$, and unsatisfiable $(3, 1, 3)$-formulas and $(3, 2, 2)$-formulas exist (which are also $(3, p, q)$-formulas for larger p, q, see Table 1). NP-hardness follows from Lemma 4. ◀

The NP-hardness part of Theorem 5 holds even for monotone SAT, where each clause is required to contain only positive or only negative literals [5], with the exception of monotone $(3, 1, 3)$-SAT, for which van Santvliet and de Haan [24] have recently shown that all instances are satisfiable, and monotone $(3, 1, 4)$-SAT, which is still open. Our proof is uniform in the sense that all hardness results rely on Lemma 4.

4 Compound methods

The core method for finding small unsatisfiable (k, s)-CNF and (k, p, q)-CNF formulas is our *SMS encoding* that we have introduced in Section 3.2. This method is quite powerful and lets us produce the smallest formulas and exact lower bounds for formula size. However, since the search space grows very quickly, this method reaches its limits with formulas of about 10–15 clauses, depending on the imposed side constraints. In this section, we show how we improve the results from Table 3 by combining computational search with theoretical analysis, and with several techniques to decrease the size of the search space. The methods for obtaining upper bounds are *disjunctive splitting* and *combining enforcers*. The methods for obtaining lower bounds are *reductions* and *hard-coding part of the adjacency matrix*.

Generating (k, s)-formulas directly is often prohibitively expensive. In such cases we also consider $(\leq k, s)$-formulas that have a few clauses of width smaller than k. Central to the various techniques we employ in this section is the concept of a stairway. A *stairway*, first introduced by Hoory and Szeider [11], is an abstraction of a CNF formula that focuses only on the clauses that are smaller than a given k. More specifically, a stairway $\sigma = (a_1, \ldots, a_r)$ is a finite non-increasing sequence of positive integers. For a fixed integer k, a stairway $\sigma = (a_1, \ldots, a_r)$ represents the set of all CNF formulas $F = \{C_1, \ldots, C_m\}$ where $a_i = k - |C_i|$ for $1 \leq i \leq r$, and $|C_i| = k$ for $r + 1 \leq i \leq m$. Define $\mu(k, s, \sigma)$ to be the number of clauses of a smallest unsatisfiable $(\leq k, s)$-formula with stairway σ, and $\mu(k, p, q, \sigma)$ to be that of the smallest unsatisfiable $(\leq k, p, q)$-formula with stairway σ.

It is straightforward to adapt $\Phi_{k,s}^{n,m}$ to encode, instead of a (k, s)-formula, a $(\leq k, s)$-formula with a given stairway. Instead of requiring every clause to contain exactly k literals, we require every clause to contain at most k literals, and that a certain number of clauses contain less than r literals, for some $1 < r \leq k$. For each $1 < r \leq k$, $j \in [n]$, let N_r be the number of clauses that contains strictly less than r literals. We replace F_5 with F_5' below.

$$F_5' := \bigwedge_{j \in [n]} \neg \tau_{j,k+1} \wedge \bigwedge_{1 < r \leq k} \sum_{j \in [n]} \neg \tau_{j,r} = N_r.$$

4.1 Improved bounds

The following theorems refine the bounds on $\mu(k, s)$ and $\mu(k, p, q)$ using stairways.

▶ **Theorem 6.** *$\mu(k,p,q) \geq \min\left(\mu(k,p,q,1^p) + q, \mu(k,p,q-1)\right)$, where 1^p is the stairway of length p with each entry being a 1.*

Proof. An unsatisfiable (k, p, q)-formula where at least one literal occurs q times gives an unsatisfiable formula with p $(k-1)$-clauses and shorter in length by q if we set the literal that occurs q times to true. The latter has minimal size $\mu(k, p, q, 1^p)$. Taking this into account, we know that $\mu(k, p, q) < \mu(k, p, q-1)$ is only possible if $\mu(k, p, q, 1^p)+q < \mu(k, p, q-1)$. ◀

Recall the concept of an enforcer from Section 2. Enforcers can be used to provide upper bounds, as we will show in Theorem 7. When $k = 3$, a $(3, s)$-enforcer (or a $(3, p, q)$-enforcer) gives rise, by appending the appropriate unit clause, to an unsatisfiable $(\leq k, s)$-formula (or an unsatisfiable $(\leq k, p, q)$-formula) with stairway (2), and thus by searching for formulas with this stairway we can generate enforcers. In this way, we computed the size of a smallest enforcer in the classes of $(3, s)$-formulas and $(3, p, q)$-formulas, and list them in Table 2. The minimality of the $(3, 3, 4)$-enforcer was also shown by Jurenka [15] with a theoretical argument. The corresponding formulas can be found in Appendix A.1.

▶ **Theorem 7.** *$\mu(k, s) \leq k \cdot (\mu(k, s,(k - 1)) - 1) + 1$. Similarly, $\mu(k, p, q) \leq k \cdot (\mu(k, p, q,(k - 1)) - 1) + 1$.*

Proof. Let E be a smallest (k, s)-enforcer, which by definition has size $\mu(k, s,(k - 1)) - 1$ (the (k, p, q) case is analogous). We can obtain an unsatisfiable (k, s)-formula by taking k variable-disjoint copies of E and adding a k-clause containing the negated enforced literals. ◀

Table 5 Smallest size of an unsatisfiable $(\leq k, 1, q)$-formula for stairway (1).

	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q \geq 5$
$p = 1$	∞	∞	14 or 15	13	11

We compute the table for $\mu(3, 1, 3,(1))$ and $\mu(3, 1, 3,(2))$ through a similar exhaustive search as explained in Section 3, and show the results in Table 5. Combining these tables with Theorems 6 and 7, we have the following improvement.

▶ **Corollary 8.** *$\mu(3, 1, 3), \mu(3, 1, 4) \geq 17$ and $\mu(3, 1, q) \geq 16$ for all $q \geq 5$.*

▶ **Corollary 9.** *$\mu(3, 1, 3) \leq 22, \mu(3, 1, 4) \leq 19$ and $\mu(3, 1, q) \leq 16$ for all $q \geq 5$.*

4.2 Disjunctive splitting

We say a CNF formula F is obtained by *disjunctive splitting in x* from CNF formulas F_1, F_2, in symbols $F = F_1 \oplus F_2$, if F can be partitioned into two nonempty sets F_1', F_2' such that the variable x occurs in F_1' positively but not negatively, and appears in F_2' negatively but not positively, and F_i is obtained from F_i' with all occurrences of $x, \overline{x}$ removed. Observe that if F_1, F_2 are unsatisfiable, then also $F = F_1' \cup F_2'$ is unsatisfiable. Hence, when constructing an unsatisfiable (k, s)-CNF or (k, p, q)-CNF formula, we can first try to construct $(\leq k, s)$-formulas or $(\leq k, p, q)$-formulas F_1, F_2 and then combine them to obtain F.

If F is obtained by disjunctive splitting in x from F_1, F_2, and x is added positively to p clauses in F_1 and negatively to q clauses in F_2, we write $F = F_1 \oplus_{p,q} F_2$. Disjunctive splitting can be recursively applied to F_1 and F_2. This allows us to construct a formula from *axioms* which are CNF formulas that we do not further split. For example, the $(2,4)$-CNF formula $\{\{x,y\}, \{\overline{x},y\}, \{\overline{y},z\}, \{\overline{y},\overline{z}\}\}$ can be constructed from the axiom $\{\square\}$.

We can describe the construction by an $\oplus$-*derivation*, an algebraic expression $(\{\square\} \oplus_{1,1} \{\square\}) \oplus_{2,2} (\{\square\} \oplus_{1,1} \{\square\})$. In fact, MU(1) is exactly the class of all formulas that can be constructed by disjunctive splitting from the axiom $\{\square\}$.

This idea was utilized by Hoory and Szeider [11], who proposed an algorithm that decides for given k, s whether (k,s)-CNF $\cap$ MU(1) $\neq \emptyset$. This allows us to compute an upper bound on the threshold function $f(k)$. Hoory and Szeider define $\oplus$-derivations to operate on stairways instead of formulas. For $k = 3$, the above $\oplus$-derivation would now read $((3) \oplus_{1,1} (3)) \oplus_{2,2} ((3) \oplus_{1,1} (3))$ and produce the stairway $(1,1,1,1)$. By means of a saturation algorithm, Hoory and Szeider could determine the upper bounds $f(3) \leq 3$, $f(4) \leq 4$, $f(5) \leq 7$, $f(6) \leq 11$, $f(7) \leq 17$, $f(8) \leq 29$, and $f(9) \leq 51$ on the threshold function $f(k)$ (i.e., all $(k, f(k))$-formulas are satisfiable but $(k, f(k)+1)$-SAT is NP-complete), which are still the best known upper bounds.

In this paper, we generalize Hoory and Szeider's method in the following ways: (i) we consider $\oplus$-derivations with more axioms: any unsatisfiable $(\leq k, s)$-formula can serve as an axiom; (ii) we modify the saturation algorithm so that it gives the size of the smallest unsatisfiable (k,s)-formula derivable with respect to the sizes of the formulas that serve as axioms; (iii) we adapt the algorithm also for searching for unsatisfiable (k,p,q)-formulas. We do this in the hope that, by finding suitable axioms with the SMS encoding, we can incorporate them into the $\oplus$-derivations to obtain smaller unsatisfiable (k,s) or (k,p,q)-formulas.

Table 6 shows the size of smallest unsatisfiable $(3,s)$ and $(3,p,q)$-formulas generated by disjunctive splitting with $\{\square\}$ being the only axiom. The corresponding $\oplus$-derivations can be found in the appendices. We determined $\mu(3,s,1^r)$ for all $1 \leq r < s$ and $\mu(3,p,q,1^r)$ for all $1 \leq r < q$, but this did not yield any new upper bounds. We were more successful with the method of disjunctive splitting in the case of $k = 4$, which we discuss in Section 5.

Table 6 Size of the smallest $(3,s)$-formulas and $(3,p,q)$-formulas in MU(1).

$s \leq 3$	$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s \geq 8$
∞	16	12	10	9	8

	$q = 1$	$q = 2$	$q = 3$	$q = 4$	$q \geq 5$
$p = 1$	∞	∞	22	19	16
$p = 2$	–	∞	12	10	10
$p = 3$	–	–	10	9	9
$p \geq 4$	–	–	–	8	8

4.3 Hard-coding part of the matrix

After we narrowed down the search scope with tighter bounds, this and the next part deal with deciding the missing values in Table 1 and the techniques involved. These techniques allow us to determine the existence of unsatisfiable $(3,4)$-formulas (or $(3,2,2)$,$(3,1,q)$-formulas) whose size is too large to be exhaustively searched by SMS directly.

In this part, we determine the value of $\mu(3,1,3)$ and $\mu(3,1,3)$. The technique here is hard-coding the part of the matrix that corresponds to the occurrences of some variables/literals. The motivation is that with less undecided values in the matrix to solve, SMS terminates more quickly. Suppose we want to fix both positive and negative occurrences of $\mathcal{V}$ variables

$A(i,2n+j)$	$1\dots$	$m-9$			$m-6$			$m-3$			m
1 (x_1)											1
2 ($\overline{x_1}$)								1	1	1	
3 (x_2)											1
4 ($\overline{x_2}$)					1	1	1				
5 (x_3)											1
6 ($\overline{x_3}$)		1	1	1							
7 (a_1)										1	
8 (b_1)										1	
9 (a_2)							1				
10 (b_2)							1				
11 (a_3)				1							
12 (b_3)				1							

Figure 2 The first rows of the matrix determined as a result of our choice for pos(i) and neg(i). The omitted values are all 0.

$x_1, x_2, \dots, x_{\mathcal{V}}$ and only the positive occurrences of $\mathcal{L}$ other variables $x_{\mathcal{V}+1}, x_{\mathcal{V}+2}, \dots, x_{\mathcal{V}+\mathcal{L}}$. To do this in a way that is compatible with the minimality check of SMS, we need to adjust the following two things. First, when generating the encoding, we stipulate that the first rows in the matrix correspond to $x_1, \overline{x}_1, x_2, \overline{x}_2, \dots, x_{\mathcal{V}}, \overline{x}_{\mathcal{V}}, x_{\mathcal{V}+1}, x_{\mathcal{V}+2}, \dots x_{\mathcal{V}+\mathcal{L}}$ and adjust pos and neg to the following.

$$\text{pos}(i) := \begin{cases} 2i-1 & \text{if } i \le \mathcal{V}, \\ i+\mathcal{V} & \text{otherwise;} \end{cases} \qquad \text{neg}(i) := \begin{cases} 2i & \text{if } i \le \mathcal{V}, \\ 2n+\mathcal{V}-i+1 & \text{otherwise.} \end{cases}$$

Second, we start SMS with the minimality check restricted to the refined partition

$$\{\{1\}, \{2\}, \dots, \{2\mathcal{V}+\mathcal{L}\}, [2\mathcal{V}+\mathcal{L}+1, 2n], [2n+1, 2n+m]\}$$

of the original $\{[1,2n], [2n+1, 2n+m]\}$. Given the specific assumption on the variables and literals fixed, the minimality condition will determine the values of the first $2\mathcal{V}+\mathcal{L}$ rows in the matrix. An example to this will be given shortly after.

To determine the value of $\mu(3,1,3)$ and $\mu(3,1,4)$, we prove the following lemma that argues about the presence of certain "partially known" $\le k$-CNFs, so we can fix them. As explained in Section 2, we write $\le k$-CNFs in matrix form. To argue about "partially known" $\le k$-CNFs, we extend the matrix notation to what is essentially a first-order language of $\le k$-CNFs. We use lowercase letters $x, y, z, a, b, c, \dots$ to denote symbols to be interpreted by propositional literals, positive or negative. We use $\overline{}$ to denote negation: if x is interpreted as some literal, then $\overline{x}$ must be interpreted as its negation. Two different literal symbols may be interpreted by different literals, or they may be interpreted by the same literal, or even by the two literals of the same variable. When a position in the matrix is left blank, we leave the corresponding literal unconstrained. We then say a formula F is *of the form* M if the symbols in the matrix M can be interpreted by the literals of F to yield a matrix of F. A clause $C \in F$ is *singular* if $\deg(x) = 1$ for all $x \in C$.

Table 7 Results of the search for unsatisfiable $(3,1,3)$-formulas for different numbers of variables and clauses after hard-coding the first rows as described. All queries were unsatisfiable except the one marked in blue with $m = 22$ and $n = 21$. For each m, the lower bound for the choice of n is by Lemma 3, and the upper bound is $m - 2$ since we know the smallest size of a $(3,1,3)$-formula in MU(1). There are two values for each pair of m and n. The values on the left indicate the time spent on the case where we assume there is a unique singular clause. The values on the right indicate that of the case where we assume there are at least two singular clauses. The cases superscribed with $\star$ and $\blacktriangle$ did not terminate within a timeout of 6 days and were further split into sub-cases and time shown is the sum of time spent on each of the sub-cases. The cases superscribed with $\star$ are split into 1258 cases in terms of where $\overline{a_1}, \overline{b_1}, \overline{a_2}$ and $\overline{a_2}$ occur, modulo symmetries. The ones superscribed with $\blacktriangle$ are split into 136 cases in terms of where $\overline{y_1}, \overline{y_2}$ and $\overline{y_3}$ occur, modulo symmetries. It is worth noting that both sets of case distinctions are generated automatically without symmetry by reformulating them as graph problems and giving the corresponding encoding to SMS. An unsatisfiable formula is found in this case, assuming there exists a unique singular clause. Upon inspection, this formula is revealed to be composed of 3 enforcers in the spirit of Theorem 7.

	$m = 17$	$m = 18$	$m = 19$	$m = 20$	$m = 21$	$m = 22$
$n = 13$	5.3s/7.4m					
$n = 14$	16s/17m	38s/1.8h				
$n = 15$	19m/1.1h	11m/28.7h	12m/43.8h	53m/13.7h*		
$n = 16$		7.2m/20h	7.4h/5.8h*	1.7d/1.2d*	1.7d/9.9d*	
$n = 17$			3.8d/9.0h*	15.9h$^\blacktriangle$/1.8d*	2.1d$^\blacktriangle$/14.3d*	
$n = 18$				1.2d$^\blacktriangle$/2.5d*	3.6d$^\blacktriangle$/34.3d*	
$n = 19$					11.6d$^\blacktriangle$/56.9d*	
$n = 21$						29.2m/–

▶ **Lemma 10** ($\star$). *Let $q \geq 3$ and let F be a smallest unsatisfiable $(3,1,q)$-formula.*

1. *There is a singular clause in F.*
2. *If $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \in F$ is singular, then $F[x_i] \cap F[x_j] = \left\{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}\right\}$ for any $i \neq j \in \{1,2,3\}$, and* $\deg(\overline{x_1}) = \deg(\overline{x_2}) = \deg(\overline{x_3}) \geq 3$.
3. *Let $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \in F$ be a singular clause and let a be a literal such that* $\deg(a) = 1$ *and* $\mathsf{var}(a) \neq \mathsf{var}(x_i)$ *for all $i \in \{1,2,3\}$. If $a \in \bigcup F[x_i]$, then $\overline{a} \notin \bigcup F[x_j]$ for any $i \neq j \in \{1,2,3\}$.*
4. *If there is a unique singular $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \in F$, then $\bigcup_{i=1,2,3} F[x_i]$ is of the form* $\begin{pmatrix} x_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_2 & \overline{x}_2 & \overline{x}_2 & \overline{x}_3 & \overline{x}_3 & \overline{x}_3 \\ x_2 & a_1 & & & a_2 & & & a_3 & & \\ x_3 & b_1 & & & b_2 & & & b_3 & & \end{pmatrix}$ *where* $\deg(a_i) = \deg(b_i) = 1$ *for all $i \in \{1,2,3\}$.*

When searching for a smallest unsatisfiable $(3,1,3)$-formula, we distinguish the following two cases, depending on whether there is a unique clause whose literals only occur once. If there is, then we hard-code the partial formula $\begin{pmatrix} x_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_2 & \overline{x}_2 & \overline{x}_2 & \overline{x}_3 & \overline{x}_3 & \overline{x}_3 \\ x_2 & a_1 & & & a_2 & & & a_3 & & \\ x_3 & b_1 & & & b_2 & & & b_3 & & \end{pmatrix}$ in the matrix by setting the first 12 rows to represent $x_1, \overline{x}_1, x_2, \overline{x}_2, x_3, \overline{x}_3, a_1, b_1, a_2, b_2, a_3$ and b_3, starting with the ordered partition $\{\{1\},\{2\},\ldots,\{12\},[13,2n],[2n+1,2n+m]\}$, and hard-coding the first 12 rows of the matrix thereby determined. The fixed rows are shown in Figure 2 as an example. Otherwise, there are more than one singular clause and we hard-code the partial formula $\begin{pmatrix} x_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_1 & \overline{x}_2 & \overline{x}_2 & \overline{x}_2 & \overline{x}_3 & \overline{x}_3 & \overline{x}_3 & y_1 \\ x_2 & & & & & & & & & & y_2 \\ x_3 & & & & & & & & & & y_3 \end{pmatrix}$. When searching for a smallest unsatisfiable $(3,1,4)$-formula, we follow the same rationale but split both of these cases into 4 cases depending on the size of $\{x_i \mid \deg^{\mathsf{var}}(x_i) = 5, i \in \{1,2,3\}\}$.

Combining the computational results in Tables 7 and 8 with the previous lemma, we have the following result.

▶ **Theorem 11.** $\mu(3,1,3) \geq 22$ *and* $\mu(3,1,4) \geq 19$.

Table 8 Results of the search for unsatisfiable $(3,1,4)$-formula for different numbers of variables and clauses after hard-coding the first rows as described. The timeout is 11 days. For each m in the table, the lower bound for the choice of n is by Lemma 3, and the upper bound is $m-2$ since we know that the smallest size of a $(3,1,4)$-formula in MU(1). There are eight values for each pair of m and n divided into two rows of four values each. The upper row shows the amount of time spent on the cases where we assume there is a unique singular clause; the lower row shows the cases with at least two singular clauses. The four values in a row correspond to cases based on the number of variables out of x_1, x_2, x_3 that are of degree 5 (0, 1, 2, 3 left-to-right). An unsatisfiable formula is found only for the case where $m = 19$, $n = 18$ and $\deg^{\mathsf{var}}(x_1) = \deg^{\mathsf{var}}(x_2) = \deg^{\mathsf{var}}(x_3) = 5$. Upon inspection, this formula is revealed to be composed of 3 enforces in the way of Theorem 7.

	$m = 17$	$m = 18$	$m = 19$
$n = 11$	(1.1s,1.1s,1.1s,0.7s)	(–,–,1.1s,0.8s)	
	(19s,15s,12s,13s)	(–,–,13s,14s)	
$n = 12$	(1.8s,2.0s,2.0s,2.0s)	(2.2s,2.5s,2.4s,2.7s)	(4.7s,2.7s,2.8s,3.0s)
	(1.1m,1.0m,0.6m,13s)	(1.2m,1.2m,0.5m,0.5m)	(2.0m,1.4m,1.9m,1.6m)
$n = 13$	(15s,7.1s,4.0s,6.4s)	(5.5s,10s,12s,8.2s)	(12s,9.0s,6.0s,14s)
	(4.5m,1.1m,0.5m,1.0m)	(6m,4.9m,0.9m,1.5m)	(11m,5.1m,10m,9.5m)
$n = 14$	(24s,43s,14s,12s)	(25s,56s,26s,32s)	(1.2m,40s,1.2m,34s)
	(9.1m,2.4m,1.2m,1.1m)	(34m,8.4m,11m,3.6m)	(1.7h,3.0h,2.8h,0.9h)
$n = 15$	(1.0m,1.7m,17s,21s)	(17m,2.7m,49s,1.8m)	(2.3m,1.0h,1.3m,9.7m)
	(42m,10m,2.3m,1.9m)	(1.7h,50m,22m,22m)	(33.2h,17.1h,2.6h,1.4h)
$n = 16$		(39m,24m,36m,3.3m)	(14.1h,4.5h,34m,30m)
		(19.7h,47m,24m,40m)	(t.o.,6.4d,3.3h,2.5h)
$n = 17$			(7.2d,12.0h,4.8h,2.6h)
			(t.o.,t.o.,10.1h,2.3h)
$n = 18$			(t.o.,5.6d,6.5h,1.1h)
			(2.4d,t.o.,6.3d,10.4h)

4.4 Reduction

In this section, we describe the final technique that allow us to determine the value of $\mu(3,4)$ and $\mu(3,2,2)$, and thus complete Table 1. The idea is to reduce a $(3,4)$-formula (or $(3,2,2)$-formula) to a smaller $(\le 3,4)$-formula (or $(\le 3,2,2)$-formula) that is equisatisfiable, so that the question of the existence of a certain formula is reduced to that of the existence of a certain, smaller formula. We then use SMS to determine the existence of such small formulas.

It is difficult to prove any useful properties about general unsatisfiable $(3,4)$-formulas (or $(3,2,2)$-formulas), but since we exhaustively search from smaller to bigger formulas, we can restrict our search to *minimal* (in terms of the number of clauses) $(3,4)$-formulas (or $(3,2,2)$-formulas). We reduce such an F to a smaller unsatisfiable $(\le 3,4)$-formula (or $(\le 3,2,2)$-formula) by replacing all subsets of clauses from F that fit into one of the three forms below with a single 2-clause $\begin{pmatrix} c \\ d \\ \times \end{pmatrix}$. For each replacement operation, the symbols c, d are instantiated separately, i.e., they could be instantiated differently each time. Each replacement is tantamount to a sequence of variable eliminations, and thus is sound (preserves unsatisfiability).

1. $\begin{pmatrix} x & \overline{x} & \overline{x} & w & \overline{w} & \overline{w} & \overline{a} & c \\ a & a & a & b & b & b & \overline{b} & d \\ z & y & \overline{y} & z & v & \overline{v} & z & \overline{z} \end{pmatrix}$ for some $\deg^{\mathsf{var}}(x) = \deg^{\mathsf{var}}(w) = 3$ and $\deg^{\mathsf{var}}(y) = \deg^{\mathsf{var}}(v) = 2$.

Elimination sequence: $\mathsf{var}(y), \mathsf{var}(v), \mathsf{var}(x), \mathsf{var}(w), \mathsf{var}(a), \mathsf{var}(b), \mathsf{var}(z)$.

2. $\begin{pmatrix} x & \overline{x} & \overline{x} & c \\ a & a & a & \overline{a} \\ d & y & \overline{y} & d \end{pmatrix}$ for some $\deg^{\mathsf{var}}(x) = 3$, $\deg^{\mathsf{var}}(y) = 2$ and d. Eliminate: $\mathsf{var}(y), \mathsf{var}(x), \mathsf{var}(a)$.
3. $\begin{pmatrix} y & \overline{y} \\ c & c \\ d & d \end{pmatrix}$ for some $\deg^{\mathsf{var}}(y) = 2$ and $\deg^{\mathsf{var}}(c) = \deg^{\mathsf{var}}(d) = 4$. Eliminate: $\mathsf{var}(y)$.

Given the number of clauses m and the number of variables n of the formula, the following lemma helps us narrow down possibilities in terms of how many subsets there are that fit into each of the forms.

▶ **Lemma 12** (⋆)**.** *Let F be an unsatisfiable $(3,4)$-formula of minimal size. Let x, y, u, w be literals such that $\mathsf{var}(y) \neq \mathsf{var}(u)$ and $\mathsf{var}(x) \neq \mathsf{var}(w)$, $\deg^{\mathsf{var}}(y) = \deg^{\mathsf{var}}(u) = 2$ and $\deg^{\mathsf{var}}(x) = \deg^{\mathsf{var}}(w) = 3$. We have the following facts:*

1. *If a literal a occurs in F, then $\overline{a}$ also occurs in F.*
2. *If $\begin{pmatrix} a \\ b \end{pmatrix} \in F$, then $\mathsf{var}(a) \neq \mathsf{var}(b)$.*
3. *$F[y]$ is of the form $\begin{pmatrix} y & \overline{y} \\ a & a \\ b & b \end{pmatrix}$.*
4. *$F[x]$ is of the form $\begin{pmatrix} x & \overline{x} & \overline{x} \\ a & a & a \\ b & c & d \end{pmatrix}$.*
5. *$F[y] \cap F[u] = \emptyset$.*
6. *$F[x] \cap F[w] = \emptyset$.*
7. *Either $F[x] \cap F[y] = \emptyset$, or $F[x]$ is of the form $\begin{pmatrix} x & \overline{x} & \overline{x} \\ a & a & a \\ z & y & \overline{y} \end{pmatrix}$.*
8. *If $F[x] = \begin{pmatrix} x & \overline{x} & \overline{x} \\ a & a & a \\ z & y & \overline{y} \end{pmatrix}$, then $F[a] = \begin{pmatrix} x & \overline{x} & \overline{x} & b \\ a & a & a & \overline{a} \\ z & y & \overline{y} & z \end{pmatrix}$, and $\deg^{\mathsf{var}}(b) = \deg^{\mathsf{var}}(z) = 4$.*

▶ **Lemma 13.** *A smallest $(3,2,2)$-formula has even size and no variables of degree* 3*.*

Proof. It is possible to prove fact 4 from Lemma 12 for a minimal $(3,2,2)$-formula as well (it does not follow automatically, as a minimal $(3,2,2)$-formula is not necessarily a minimal $(3,4)$-formula, but the proof is based on the same idea). A variable x of degree 3 then implies there is a literal a of degree 3; a contradiction in a $(3,2,2)$-formula. So, a minimal $(3,2,2)$-formula contains only variables of type $(1,1)$ and $(2,2)$, and has an even number of literal occurrences. With $k = 3$, the number of clauses must be even. ◀

One caveat to searching for a formula of a reduced profile using SMS is that the reduction can reduce degrees of some variables and literals. This means that some of the variables or literals in the shorter clauses must occur strictly fewer times than the general bound s (or p, q). To accommodate for this, we count each literal that occurs in a 2-clause twice. However, this means that the literals in the 2-clauses whose degree was not reduced by the steps above may exceed their degree cap under this way of counting. To adjust to this, we sum up the number of exceeding counts and call it the *surplus* of the formula. Given a specific reduction, we know the exact value of allowed surplus for the reduced formula, and so we can include it as a part of the constraints. Here as an example, we write out the formula for the case of bounded variable degree. The formula for the literal case can be defined similarly. For all $i \in [n] \cup [-n], j \in [m]$, define $e'_{i,j} := e_{i,j} \wedge \neg\tau_{j,k}$. For all $i \in [m]$ and $t \in [m] \cup [-m]$, define $(\sigma'_{i,1}, \sigma'_{i,2}, \ldots, \sigma'_{i,s+s}) := \mathrm{Card}(\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,s}, e'_{i,1}, \ldots, e'_{i,m}, e'_{-i,1}, \ldots, e'_{-i,m})$. Suppose $\mathcal{S}$ is the value of surplus. Then we can define

$$F_{\text{surp}} := \sum_{i \in [n]} \sigma'_{i,s+1} + \cdots + \sigma'_{i,s+s} = \mathcal{S}.$$

We call the combination of the number of variables n, the number of clauses m, a stairway σ and the number of surplus $\mathcal{S}$ a *profile*. For further speed-up, we also hard-code a

Table 9 Number $\mathcal{V}$ of hard-coded variables of degree 3 we chose and time spent determining the existence of a formula with each profile from reduction for previously unsolved cases with n variables and m clauses from Table 4. Naturally, in each case $\mathcal{V}$ is no greater than the total number of variables of degree 3 after reduction. In the table on the right, the number of hard-coded variables of degree 3 is always 0 because by Corollary 13 no variable of degree 3 exists.

n	m	n'	m'	#2-cl	$\mathcal{S}$	$\mathcal{V}$	time
12	14	9	11	1	1	3	0.2s
		10	12	2	0	2	2.0s
		8	10	2	1	0	32s
		5	7	1	2	0	0.3s
		6	8	2	2	0	1.5s
		9	11	3	0	0	62s
12	15	12	15	0	0	3	6.5s
		11	14	1	0	1	2.1h
		9	12	1	1	0	41m
13	15	11	13	2	0	3	0.7s
		9	11	2	1	2	1.3s
		6	8	1	2	0	1.0s
		7	9	2	2	0	9.4s
		10	13	3	0	1	1.6h
		8	10	3	1	0	33s

n	m	n'	m'	#2-cl	$\mathcal{S}$	time
12	16	12	16	0	0	5d
13	16	11	14	2	0	7m
14	16	10	12	4	0	5.2s
14	18	13	17	1	0	21d
15	18	12	15	3	0	25m
16	18	11	13	5	0	4.5s

number of variables of degree 3 in the same way as described in the previous part. Let F be a smallest unsatisfiable $(3,4)$-formula and let $x_1, x_2, \dots, x_{\mathcal{V}}$ be $\mathcal{V}$ degree 3 variables in F whose occurrences we want to fix. By Lemma 12, the formula is of the following form. We fix the first $3\mathcal{V}$ rows of the matrix thereby determined.

$$\begin{pmatrix} x_1 & \overline{x}_1 & \overline{x}_1 \\ a_1 & a_1 & a_1 \end{pmatrix}\begin{pmatrix} x_2 & \overline{x}_2 & \overline{x}_2 \\ a_2 & a_2 & a_2 \end{pmatrix}\dots\begin{pmatrix} x_{\mathcal{V}} & \overline{x}_{\mathcal{V}} & \overline{x}_{\mathcal{V}} \\ a_{\mathcal{V}} & a_{\mathcal{V}} & a_{\mathcal{V}} \end{pmatrix}\dots$$

▶ **Lemma 14.** *A smallest unsatisfiable* $(3,4)$*-formula (or* $(3,2,2)$*-formula) with* m *variables and* n *clauses exists only if a formula of one of the profiles in Table 9, left (right) exists.*

Proof. We enumerate all possible numbers of variables of degree 2 and 3 in the (unreduced) formula, and list all possibilities of how the clauses of variables of degree 2 and 3 overlap according to Lemma 12. We then perform the reduction to each possibility and obtain a list of profiles that the unsolved cases from Table 9 reduce to. ◀

We tested the existence of each of the profiles from Table 9 with SMS and all of them returned negative. Combining these experiment results with Lemma 14, we obtain the following theorem.

▶ **Theorem 15.** $\mu(3,4) > 15$ *and* $\mu(3,2,2) > 18$.

5 A smaller unsatisfiable $(4,5)$-formula

We now turn our attention to unsatisfiable CNF formulas with clauses of length 4. As mentioned in the introduction, $f(4) = 4$, i.e., $s = 5$ is the smallest value so that an unsatisfiable $(4,s)$-formula exists. We found an unsatisfiable $(4,s)$-formula with 235 clauses, improving upon the formulas provided by Stříbrná [26] (449 clauses) and Knuth [20, p. 588] (257 clauses). Knuth's formula $K_{4,5}$ can be described by the $\oplus$-derivation from the axiom $\{\square\}$ depicted as a directed acyclic graph in Figure 3.

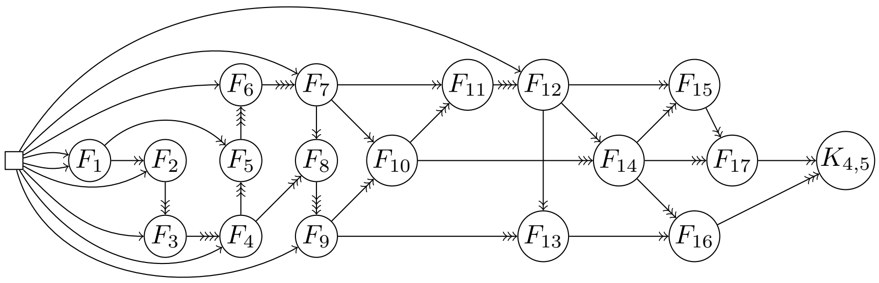


Figure 3 The $\oplus$-derivation of Knuth's formula $K_{4,5}$. Each node has two incoming arcs; the number of arrowheads denotes the values p, q in $\oplus_{p,q}$.

This formula has 257 clauses and 256 variables. However, 22 of these variables occur twice (introduced in F_1) and 31 variables occur in three clauses (introduced in F_2). Hence we can identify each of the variables with 2 occurrences with one variable with 3 occurrences, still keeping the number of occurrences of the new identified variable within the bound 5. By this method, we can save 22 variables, and indeed, Knuth states his formula to have $256 - 22 = 234$ variables. With our modified saturation algorithm we could show that $K_{4,5}$ is a smallest unsatisfiable $(4,5)$-formula in MU(1). Thus, for finding a smaller formula, one needs to search outside MU(1), and outside the class of formulas that can be obtained from an MU(1) formula by identifying pairs of low-occurrence variables.

Finding an entire unsatisfiable $(4,5)$-formula with SMS does not seem feasible. However, we can search for an unsatisfiable $(\leq 4,5)$-formula F that represents a stairway σ with fewer clauses than any formula in MU(1) that represents the same stairway σ. Thus way we can use F as an additional axiom in $\oplus$-derivations and this way possibly find a smaller unsatisfiable $(4,5)$-formula. We considered all stairways $\sigma \in \{(3), (2), (3,2), (2,2), (2,2,2), (1), (3,1), (2,1), (2,2,1), (1,1), (3,1,1), (2,1,1), (2,2,1,1), (1,1,1), (3,1,1,1), (2,1,1,1), (1,1,1,1)\}$ and run our SMS encoding to find an unsatisfiable formula F with at most 13 clauses and fewer clauses than a smallest MU(1)-formula for σ.

The search resulted in two such formulas within a timeout of five days. One formula has 7 clauses for the stairway $(3,1,1,1)$ and the other has 8 clauses for the stairway $(3,2)$; shortest MU(1) formulas for these stairways have 8 and 9 clauses, respectively. Using the first of these two formulas as axiom indeed reduces the size of the unsatisfiable $(4,5)$-formula from 257 to 235, since the axiom is used several times; the second formula can be derived by an $\oplus$ operation from the new axiom and axiom $\{\square\}$. Our smaller unsatisfiable $(4,5)$-formula can be obtained by replacing F_6 with F_6' in the $\oplus$-derivation from Figure 3, and using F_6' as an additional axiom, where the two formulas are as follows (rows are variables, columns are clauses, $+/-$ indicates positive/negative occurrence; c.f. the appendix).

$$F_6' = \begin{pmatrix} - & & & & & & + \\ & - & - & - & - & + & \\ - & & - & + & + & & - \\ - & - & + & & + & & - \\ - & + & & - & + & & - \end{pmatrix} \qquad F_6 = \begin{pmatrix} & & & - & - & - & - & + \\ & & - & - & - & + & + & \\ - & - & - & - & + & & & \\ - & - & - & + & & & & \\ - & - & + & & & & & \\ - & + & & & & - & + & \end{pmatrix}$$

6 Conclusion

We have identified the smallest unsatisfiable $(3, s)$ and $(3, p, q)$-formulas for a comprehensive range of values, and brought an improvement in the known minimal size for an unsatisfiable $(4, 5)$-CNF formula. Our work also contributed a uniform proof of the dichotomy for $(3, p, q)$-SAT. The core methodology, a fusion of theoretical insights and an innovative application of the SMS framework with the methods of disjunctive splitting and reductions has not only led to the discovery of new smallest unsatisfiable formulas but also demonstrated the practical utility of SMS in exploring the combinatorial landscape of CNF formulas.

Our findings have illuminated several challenging and open avenues for future work. For instance, extending the scope to identify smallest unsatisfiable formulas for $k \geq 4$ remains a significant challenge. While the methods developed here provide a solid foundation, both novel techniques and theoretical advances are necessary to tackle the increased complexity of larger k values. An extension of our methods may lead to determining the exact value of the threshold $f(5)$, currently only known to be in the interval $[5, 7]$. Moreover, the interplay between the size of unsatisfiable formulas and their implications for inapproximability results in MaxSAT problems [3] invites deeper investigation. Finally, since we found out that, for all $q \geq 3$, the size of the smallest unsatisfiable $(3, 1, q)$-formula coincide with that of the smallest unsatisfiable $(3, 1, q)$-formula in MU(1), we conjecture that this is true for all $k \geq 3$ and $q \geq f(k) + 1$.

References

1 Ron Aharoni and Nathan Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *J. Combin. Theory Ser. A*, 43:196–204, 1986. doi:10.1016/0097-3165(86)90060-9.

2 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009. doi:10.1007/978-3-642-02777-2_18.

3 Piotr Berman, Marek Karpinski, and Alex D. Scott. Approximation hardness and satisfiability of bounded occurrence instances of SAT. Technical Report TR03-022, *Electronic Colloquium on Computational Complexity* (ECCC), 2003. URL: https://eccc.weizmann.ac.il/report/2003/022/.

4 Piotr Berman, Marek Karpinski, and Alexander Scott. Approximation hardness of short symmetric instances of max-3sat. Technical Report TR03–049, *Electronic Colloquium on Computational Complexity* (ECCC), 2003. URL: https://eccc.weizmann.ac.il/report/2003/049/.

5 Andreas Darmann and Janosch Döcker. On simplified np-complete variants of monotone3-sat. *Discret. Appl. Math.*, 292:45–58, 2021. doi:10.1016/J.DAM.2020.12.010.

6 M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of the ACM*, 7(3):201–215, 1960. doi:10.1145/321033.321034.

7 Olivier Dubois. On the r, s-SAT satisfiability problem and a conjecture of Tovey. *Discr. Appl. Math.*, 26(1):51–60, 1990. doi:10.1016/0166-218X(90)90020-D.

8 Johannes K. Fichte, Markus Hecher, Daniel Le Berre, and Stefan Szeider. The silent (r)evolution of SAT. *Communications of the ACM*, 66(6):64–72, June 2023. doi:10.1145/3560469.

9 Heidi Gebauer. Disproof of the neighborhood conjecture with implications to SAT. *Combinatorica*, 32(5):573–587, 2012. doi:10.1007/S00493-012-2679-Y.

10 Heidi Gebauer, Tibor Szabó, and Gábor Tardos. The local lemma is asymptotically tight for SAT. *J. of the ACM*, 63(5):Art. 43, 32, 2016. doi:10.1145/2975386.

11 Shlomo Hoory and Stefan Szeider. Computing unsatisfiable k-SAT instances with few occurrences per variable. *Theoretical Computer Science*, 337(1-3):347–359, 2005. doi:10.1016/j.tcs.2005.02.004.

12 Shlomo Hoory and Stefan Szeider. A note on of unsatisfiable k-CNF formulas with few occurrences per variable. *SIAM J. Discrete Math.*, 20(2):523–528, 2006. doi:10.1137/S0895480104445745.

13 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. doi:10.1007/978-3-642-12002-2_10.

14 Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf QBF solving with QCIR. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop.*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016. URL: https://aaai.org/papers/aaaiw-ws0186-16-12601/.

15 David Jurenka. Upper bounds for (k, s)-SAT. Bachelor's Thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011. URL: http://hdl.handle.net/20.500.11956/50583.

16 Markus Kirchweger, Tomáš Peitl, and Stefan Szeider. Co-certificate learning with SAT modulo symmetries. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 1944–1953. ijcai.org, 2023. Main Track. doi:10.24963/IJCAI.2023/216.

17 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, LIPIcs, pages 39:1–39:17. Dagstuhl, 2021. doi:10.4230/LIPIcs.CP.2021.34.

18 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Transactions on Computational Logic*, 2024. Full and extended version of [17], to appear.

19 Hans Kleine Büning and Theodor Lettman. *Propositional logic: deduction and algorithms.* Cambridge University Press, Cambridge, 1999.

20 Donald E. Knuth. *The art of computer programming. Vol. 4B. Combinatorial algorithms. Part 2.* Addison-Wesley, Upper Saddle River, NJ, 2023.

21 Jan Kratochvíl, Petr Savický, and Zsolt Tuza. One more occurrence of variables make satisfiability jump from trivial to NP-complete. *SIAM J. Comput.*, 30:397–403, 1993. doi:10.1137/0222015.

22 O. Kullmann. On a generalization of extended resolution. *Discrete Appl. Math.*, 96–97(1):149–176, October 1999. doi:10.1016/S0166-218X(99)00037-2.

23 João P. Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

24 Hannah Van Santvliet and Ronald de Haan. All instances of monotone 3-sat-(3,1) are satisfiable, 2023. arXiv:2311.06563.

25 P. Savický and Jiří Sgall. DNF tautologies with a limited number of occurrences of every variable. *Theoretical Computer Science*, 238(1-2):495–498, 2000. doi:10.1016/S0304-3975(00)00036-0.

26 J. Stříbrná. Between combinatorics and formal logic. Master's thesis, Charles University, Prague, 1994.

27 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discr. Appl. Math.*, 8(1):85–89, 1984. doi:10.1016/0166-218X(84)90081-7.

28 Tianwei Zhang, Tomáš Peitl, and Stefan Szeider. Small unsatisfiable k-cnfs with bounded literal occurrence, 2024. arXiv:2405.16149.

A Formulas

In the appendix we list formulas in tabular form, where rows represent variables, columns represent clauses, and the $+$ and $-$ signs indicate positive and negative occurrence, respectively. Blank cells mean the variable does not occur in the clause.

A.1 Smallest enforcers

In the following, we give examples of smallest $(3, s)$-enforcers $E_{3,s}$ and smallest $(3, p, q)$-enforcers $E_{3,p,q}$ for all entries in Table 2. Note that $E_{3,4}$ also acts as a smallest $(3, 2, 3)$-enforcer.

$$E_{3,5} = \begin{pmatrix} - & - & - & - \\ - & - & + & + \\ - & + & - & + \end{pmatrix} \qquad E_{3,4} = \begin{pmatrix} & - & - & - & \\ - & & - & + & + \\ - & - & + & & + \\ - & + & & - & + \end{pmatrix} \qquad E_{3,1,5} = \begin{pmatrix} - & - & - & - & - \\ - & & & - & + \\ & - & - & & + \\ - & & & + & \\ & - & + & & \end{pmatrix}$$

$$E_{3,1,4} = \begin{pmatrix} - & & & & - & + \\ & - & - & - & & + \\ - & & & - & - & - \\ - & & & & + & \\ & - & - & + & & \\ & - & + & & & \end{pmatrix} \qquad E_{3,1,3} = \begin{pmatrix} & & & & + & + & + \\ - & & & - & & - & + \\ & - & - & & - & & + \\ - & & & - & & + & \\ & - & - & & + & & \\ - & & & + & & & \\ & - & + & & & & \end{pmatrix} \qquad E_{3,2,2} = \begin{pmatrix} - & & - & & & & & & & \\ & - & & & & - & & & + & + \\ & & & & - & & - & + & & + \\ - & & & - & & & + & & & + \\ & & & & - & + & & - & + & \\ & - & - & & & + & & + & & \\ & & & - & + & & - & & + & \\ - & - & + & + & & & & & & \end{pmatrix}$$

A.2 Smallest MU(1) formulas

In the following, we give $\oplus$-derivations of the smallest $(3, s)$-formula $M^1_{3,s}$ and the smallest $(3, p, q)$-formula $M^1_{3,p,q}$ restricted to MU(1) for each entry from Table 6. For all the derivations, we have $F_1 = \{\square\} \oplus_{1,1} \{\square\}$, $F_2 = \{\square\} \oplus_{1,2} F_1$ and $F_{2'} = F_1 \oplus_{2,2} F_1$; other F_i symbols are defined locally in each derivation.

$M^1_{3,4} = F_4 \oplus_{2,2} F_4$	$F_3 = \{\square\} \oplus_{1,3} F_2$	$F_4 = F_3 \oplus_{2,2} F_3$		
$M^1_{3,5} = F_4 \oplus_{3,2} F_5$	$F_3 = \{\square\} \oplus_{1,3} F_2$	$F_4 = F_1 \oplus_{2,3} F_2$	$F_5 = F_2 \oplus_{3,2} F_3$	
$M^1_{3,6} = F_{2'} \oplus_{4,2} F_3$	$F_3 = F_1 \oplus_{2,4} F_{2'}$			
$M^1_{3,7} = F_{2'} \oplus_{4,3} F_3$	$F_3 = F_1 \oplus_{2,3} F_2$			
$M^1_{3,8} = F_{2'} \oplus_{4,4} F_{2'}$				
$M^1_{3,1,3} = F_6 \oplus_{1,3} F_4$	$F_3 = \{\square\} \oplus_{1,3} F_2$	$F_4 = F_3 \oplus_{1,3} F_2$	$F_5 = \{\square\} \oplus_{1,3} F_4$	$F_6 = F_5 \oplus_{1,3} F_4$
$M^1_{3,1,4} = F_5 \oplus_{1,3} F_3$	$F_3 = F_2 \oplus_{1,3} F_2$	$F_4 = \{\square\} \oplus_{1,4} F_3$	$F_5 = F_4 \oplus_{1,3} F_3$	
$M^1_{3,1,5} = F_5 \oplus_{1,5} F_3$	$F_3 = F_1 \oplus_{1,3} F_2$	$F_4 = \{\square\} \oplus_{1,5} F_3$	$F_5 = F_4 \oplus_{1,5} F_3$	
$M^1_{3,2,3} = F_5 \oplus_{2,3} F_4$	$F_3 = \{\square\} \oplus_{1,3} F_2$	$F_4 = F_1 \oplus_{2,3} F_2$	$F_5 = F_3 \oplus_{2,3} F_2$	
$M^1_{3,2,4} = F_3 \oplus_{2,4} F_{2'}$	$F_3 = F_1 \oplus_{2,4} F_{2'}$			
$M^1_{3,3,3} = F_3 \oplus_{3,3} F_3$	$F_3 = F_1 \oplus_{2,3} F_2$			
$M^1_{3,3,4} = F_3 \oplus_{3,4} F_{2'}$	$F_3 = F_1 \oplus_{2,3} F_2$			
$M^1_{3,4,4} = F_{2'} \oplus_{4,4} F_{2'}$				

A.3 Smallest $(3, s)$ and $(3, p, q)$-formulas

In the following, we give examples of smallest $(3, s)$-formulas $M_{3,s}$ and smallest $(3, p, q)$-formulas $M_{3,p,q}$ for all entries in Table 1. We take $M_{3,4} = M^1_{3,4}$ and $M_{3,1,q} = M^1_{3,1,q}$ for all applicable q.

$$M_{3,6} = M_{3,3,3} = \begin{pmatrix} & - & & - & - & + & + & + \\ - & & - & & + & - & + & + \\ - & - & + & + & & - & & + \\ - & + & - & + & - & + & & \end{pmatrix}$$

$$M_{3,5} = M_{3,2,3} = \begin{pmatrix} & & - & & - & & & - & & + & + \\ & - & & - & & & - & & + & & + \\ - & & & & & - & & + & - & & + \\ - & - & & - & & & + & & & + & \\ - & & & & & - & + & & + & - & \\ & & - & & - & + & & + & & & \\ & - & - & + & + & & & & & & \end{pmatrix}$$

$$M_{3,2,4} = \begin{pmatrix} & & & & - & & & & & + \\ & & & - & & & & & + & \\ & & - & & & & & + & & \\ & - & & & & & + & & & \\ - & & & & & + & & & & \\ - & & & - & + & - & & & - & + \\ & - & - & & + & & - & - & & + \\ - & & & + & & - & & & + & \\ & - & + & & & & - & + & & \end{pmatrix}$$

$$M_{3,2,2} = \begin{pmatrix} + & & & & & + & & & & & - & & & & & - & & & & \\ & + & + & & & & - & - & & & & & & & & & & & & \\ & & & & & & & & & & & + & + & & & & - & - & & \\ & & & & & & & & & & & & & & & - & & + & + & - \\ & & & & & & & & & & & & & & & + & - & & + & - \\ & & & & & & & & & & & & & & & & + & - & + & - \\ & & & & & & & & & & - & & + & + & - & & & & & \\ & & & & & & & & & & + & - & & + & - & & & & & \\ & & & & & & & & & & & + & - & + & - & & & & & \\ & & & & & - & & + & + & - & & & & & & & & & & \\ & & & & & + & - & & + & - & & & & & & & & & & \\ & & & & & & + & - & + & - & & & & & & & & & & \\ - & & + & + & - & & & & & & & & & & & & & & & \\ + & - & & + & - & & & & & & & & & & & & & & & \\ & + & - & + & - & & & & & & & & & & & & & & & \end{pmatrix}$$